TREBALL DE FI DE CARRERA

TÍTOL: Estudi i implementació d'una aplicació de localització en xarxes de sensors

AUTOR: Ignacio Javier Borraz Gómez

DIRECTOR: Carles Gómez Montenegro

DATA: 25 de febrer de 2005

Títol: Estudi i implementació d'una aplicació de localització en xarxes de sensors

Autor: Ignacio Javier Borraz Gómez

Titulació: Enginyeria Tècnica de Telecomunicació

Especialitat: Telemàtica

Pla: 2000

Director: Carles Gómez Montenegro

Departament: Enginyeria Telemàtica

Vist i plau,

Director del TFC

Registre:

Títol: Estudi i implementació d'una aplicació de localització en xarxes de sensors

Autor: Ignacio Javier Borraz Gómez

Director: Carles Gómez Montenegro

Data: 25 de febrer de 2005

Resum

En septiembre de 1999, la revista *Business Week* predecía que la tecnología en redes de sensores iba a ser una de las 21 tecnologías más importantes para el siglo 21.
Dispositivos ligeros y baratos con múltiples sensores integrados, comunicados en red mediante enlaces inalámbricos, con acceso a Internet y desplegados en gran número, proporcionan oportunidades sin precedentes en control e instrumentación de hogares, ciudades e incluso del medio ambiente.

Para muchas de las aplicaciones potenciales de las redes de sensores es requisito imprescindible el conocimiento de la localización, bien por el propio dispositivo sensor, bien por el equipo que administre la red.
Los actuales sistemas de posicionamiento global no son suficiente para cumplir este objetivo, su utilización requeriría equipar a cada dispositivo sensor con un hardware específico y costoso y le supondría un alto gasto energético. Aunque algunos de los mecanismos de localización propuestos cuentan con la recepción de la señal GPS, limitan la capacidad receptora a unos pocos dispositivos de referencia. Debido a esto, la necesidad de desarrollar sistemas de localización especialmente diseñados para redes de sensores es evidente.

Este proyecto pretende estudiar la viabilidad de implementar un sistema de localización para redes de sensores basado en el tiempo de vuelo de señales sonoras y, en caso afirmativo, implementar una aplicación de localización que permita su testeo en un escenario de prueba.

Sus objetivos son tres. Estudiar los requisitos de un sistema de localización basado en el tiempo de vuelo de señales sonoras. Estudiar si los dispositivos sensores de los que se dispone y su sistema operativo permiten cumplir los requisitos. Diseñar un mecanismo de localización e implementarlo en una aplicación para valorar su funcionamiento en un escenario de prueba.

Abstract

In September 1999, *Business Week* magazine heralded the technology of microsensor networks as one of the 21 most important technologies for the 21st century.

Cheap, smart devices with multiple onboard sensors, networked through wireless links and the Internet and deployed in large numbers, provide unprecedented opportunities for instrumenting and controlling homes, cities, and the environment.

Knowledge of location is an indispensable requirement for many potential applications of sensor networks, either using own sensor devices or through the network administrative equip**.**

Existing global positioning systems are not enough to achieve this goal**.** Their utilization would require equipping every sensor device with specific and expensive hardware and would suppose high energy consumption. Although some of the proposed location systems rely on the GPS signal reception, these limit the reception capability to a few reference devices. For this reason, the need to develop specially designed location systems for sensor networks is evident.

This project proposes to study the possibility of creating a sensor network location system based on sound signal time-of-flight and, if results are affirmative, to construct a location application that will allow testing under experimental conditions.

The project has three main goals. To study the requirements for a sound signal based location system. To study if the available sensor devices and their operating system satisfy those requirements. To design a location system and evaluate its operation in a test environment.

DEDICATORIA

Quiero dedicar este TFC a mi director de proyecto, Carles Gómez, por la paciencia que ha tenido conmigo en algunos momentos y por haberme dado la oportunidad de trabajar en algo tan fascinante como las redes de sensores,

a Sergi, porque sin él yo nunca habría estado aquí ni seguramente llegado tan lejos,

a Ricard, Dani, Raúl y Xavi, por ayudarme cada fin de semana a olvidar las preocupaciones con carcajadas y brindis,

a Miguel, que en tantos laboratorios tuvo que sufrirme durante estos años de estudio,

a aquellos profesores que siempre han recibido con la puerta abierta a mis inquietudes y a mis ganas de aprender más,

a José Ramón, Marcos y Rubén por ser capaces de hacer del trabajo en grupo un nexo de unión y compañerismo,

a Natalia, porque ella ha sido mi fuerza en mi debilidad, mi ilusión en mi resignación y mi tranquilidad en mi desespero,

y sobretodo,

sobretodo,

a mi madre, porque ojalá hubiese podido estar aquí para verlo.

ÍNDICE

INTRODUCCIÓN

En septiembre de 1999, la revista *Business Week* predecía que la tecnología en redes de sensores iba a ser una de las 21 tecnologías más importantes para el siglo 21.

Dispositivos ligeros y baratos con múltiples sensores integrados, comunicados en red mediante enlaces inalámbricos, con acceso a Internet y desplegados en gran número, proporcionan oportunidades sin precedentes en control e instrumentación de hogares, ciudades e incluso del medio ambiente.
Por añadidura, las redes de sensores proporcionan la tecnología para un amplio abanico de sistemas en el campo de la defensa, generando nuevas capacidades potenciales en reconocimiento y vigilancia así como en otras aplicaciones tácticas.

Microsensores desechables y ligeros pueden ser desplegados en el suelo, en el aire, debajo del agua, en cuerpos, en vehículos y en el interior de edificios. Un sistema de sensores en red puede detectar y realizar el seguimiento de vehículos terrestres y alados, personal, agentes químicos y biológicos, etc y ser utilizado para fijar el emplazamiento de blancos y para el control de acceso en áreas restringidas.

Cada nodo sensor tiene la capacidad de procesamiento empotrado, y puede tener múltiples sensores integrados, operando en modos como el acústico, sísmico, infrarrojo (IR), magnético e incluso en modo de captura de imágenes y radar.
También almacenan la información sobre los enlaces inalámbricos hacia los nodos vecinos, y el conocimiento de localización y posicionamiento a través del sistema de posicionamiento global (GPS) o bien a través de algoritmos de posicionamiento local, ya que para muchas de las aplicaciones potenciales de las redes de sensores es requisito imprescindible el conocimiento de la localización.

Los actuales sistemas de posicionamiento global no son suficiente para cumplir este objetivo, su utilización requeriría equipar a cada dispositivo sensor con un hardware específico y costoso y le supondría un alto gasto energético. Aunque algunos de los mecanismos de localización propuestos cuentan con la recepción de la señal GPS, limitan la capacidad receptora a unos pocos dispositivos de referencia. Debido a esto, la necesidad de desarrollar sistemas de localización especialmente diseñados para redes de sensores es evidente.

Este proyecto pretende estudiar la viabilidad de implementar un sistema de localización para redes de sensores basado en el tiempo de vuelo de señales sonoras y, en caso afirmativo, implementar una aplicación de localización que permita su testeo en un escenario de prueba.

Sus objetivos son tres. Estudiar los requisitos de un sistema de localización basado en el tiempo de vuelo de señales sonoras. Estudiar si los dispositivos sensores de los que se dispone y su sistema operativo permiten cumplir los requisitos. Diseñar un mecanismo de localización e implementarlo en una aplicación para valorar su funcionamiento en un escenario de prueba.

La primera parte de esta memoria presenta los fundamentos teóricos necesarios para el desarrollo de este proyecto.
En el primer capítulo se hace una visión general de las redes de sensores, se explica brevemente su historia, se explican sus características básicas centrándose ya en lo que son las redes de microsensores inalámbricas y finalmente se realiza una comparación para evidenciar sus profundas diferencias con las redes inalámbricas tradicionales.
El segundo trata sobre los sistemas y mecanismos de localización, el otro eje temático de este TFC. Allí se describen las características de los mecanismos de localización, haciendo hincapié en las técnicas de cálculo de posición principales. Después se presenta brevemente el funcionamiento de los sistemas de posicionamiento global y finalmente se mencionan y explican algunos de los mecanismos que ya se han propuesto para redes de sensores.

La segunda parte de esta memoria presenta el entorno de trabajo y sus herramientas principales. En el tercer capítulo se introduce un nuevo sistema operativo TinyOS, diseñado especialmente para sensores y el lenguaje nesC en el que está programado el sistema y en el que se programan las aplicaciones para dispositivos sensores. Por último se presentan los dispositivos sensores con los que se ha trabajado.

La tercera parte se centra en lo que ha sido el desarrollo de este proyecto en sí. Se parte de la idea general para introducirse en los entresijos de la estructura de interfaces y componentes que componen las librerías y las aplicaciones de TinyOS a medida que se implementan, se descartan y se replantean los requisitos que el mecanismo de localización debe cumplir.
Más adelante, en el quinto capítulo se detallan las pruebas realizadas para testear el mecanismo de localización desarrollado y se muestran los resultados obtenidos en ellas.

Por último, en función de los resultados se extraen conclusiones sobre los objetivos alcanzados en el proyecto y se proponen vías futuras de desarrollo para posteriores proyectos.

Se han añadido cuatro anexos. El primero de ellos profundiza en las redes de sensores y en los requisitos necesarios en cada capa de comunicación. Los dos siguientes están dedicados al código desarrollado y hay extensos comentarios para facilitar su comprensión, tanto de las ideas de diseño como de la implementación en sí. Y por último, a modo de curiosidad, se han añadido los correos intercambiados en la lista de correo de desarrolladores de TinyOS.

CAPÍTULO 1: REDES DE SENSORES

1.1. Historia de las redes de sensores

Como sucede con muchas tecnologías, el desarrollo de las redes de sensores nació en el seno de la investigación para aplicaciones militares.
Dos claros ejemplos de ello, fueron el sistema SOSUS (Sistema de Vigilancia Sónico) a principios de los 50, en el cual sensores acústicos desplegados en el fondo del océano detectaban y seguían el rastro a los submarinos soviéticos, y una red de radares de defensa aérea; ambos desarrollados en el marco de la guerra fría.
Posteriormente se aplicó esta tecnología para aplicaciones civiles como fueron redes de radares para control del tráfico aéreo o el despliegue sensor de la red eléctrica nacional.
Estas primeras redes de sensores generalmente adoptaban una estructura de procesado jerárquico donde el procesado ocurría en niveles consecutivos antes de que la información sobre los eventos de interés llegase al usuario. Los nodos eran grandes estaciones distantes espacialmente y su comunicación tenía lugar a través de una infraestructura cableada.

El nacimiento de la investigación moderna podría atribuirse al programa DSN (Distributed Sensor Networks) de la DARPA, en que se pretendía comprobar si el método de comunicación de la recién aparecida Arpanet era extensible a las redes de sensores. La red de pruebas estaba compuesta por muchos nodos sensores de bajo coste distribuidos espacialmente. Los nodos colaboraban unos con otros pero operaban de forma autónoma, y la información se enrutaba hacia cualquier nodo que pudiese hacer el mejor uso de la información**.**
Aunque los primeros investigadores en redes de sensores tenían en mente redes compuestas por un gran número de pequeños sensores, la tecnología para pequeños sensores todavía no estaba suficientemente desarrollada.

En las décadas de los 80 y los 90, este tipo de redes se convirtieron en un componente crucial de los sistemas militares. Las redes de sensores perfeccionaban el desempeño de la detección y el rastreo a través de múltiples observaciones, aprovechando su amplio rango de detección y su pequeño tiempo de respuesta.

Avances recientes en computación y comunicaciones han causado un cambio significativo en la investigación en redes de sensores, acercándola hacia la consecución de la visión original. Baratos y diminutos sensores basados en la tecnología de sistemas microelectromecánicos (MEMS), la comunicación inalámbrica y procesadores baratos de bajo consumo, permiten el despliegue de redes inalámbricas ad-hoc para varias aplicaciones.
El recientemente concluido programa SensIT (Sensor Information Technology) de la DARPA perseguía dos objetivos clave en investigación y desarrollo.
El primero, el desarrollo de nuevas técnicas de operación en red apropiadas para entornos ad-hoc muy dinámicos.
El segundo objetivo era el procesado de información en red, es decir, como extraer información actualizada, útil y fiable desde la red de sensores desplegada.

1.2. Características de las redes de sensores actuales

El desarrollo de las redes de sensores requiere tecnologías de tres áreas de investigación diferentes: detección, comunicación, y computación (incluyendo hardware, software y algoritmia).

Los nodos sensores se encuentran normalmente esparcidos en un campo sensor (ver figura 1.1). Cada uno de estos nodos sensores esparcidos tiene las capacidades de recolectar datos y enrutarlos hacia el nodo recolector (del inglés *sink* cuya traducción literal es "sumidero") mediante una arquitectura ad-hoc (sin infraestructura) de múltiples saltos.
El nodo recolector puede comunicar con el nodo administrador (gestor de tareas) vía internet o vía satélite.

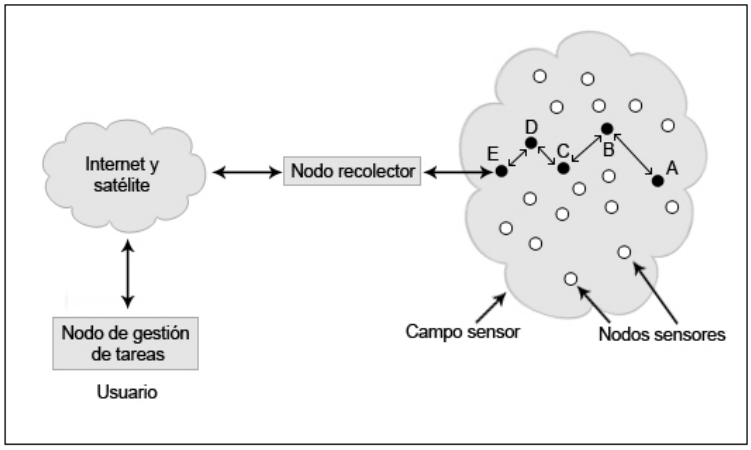


Fig.1.1 Estructura de una red sensora

El diseño de una red sensora como la descrita aquí está influenciado por los siguientes factores:

1.2.1. Tolerancia a fallos

Algunos nodos sensores pueden fallar o bloquearse debido a la falta de energía, o recibir daños físicos o interferencias medioambientales. El fallo de nodos sensores no debería comprometer el funcionamiento global de la red sensora. Este es el principio de la tolerancia a fallos o fiabilidad.
La fiabilidad $R_k(t)$ o tolerancia a fallos de un nodo sensor está modelado en [3] utilizando una distribución de Poisson para capturar la probabilidad de no tener un fallo durante un determinado intervalo de tiempo $(0,t)$:

$$R_k(t) = e^{-\lambda_k t} \qquad \textbf{(1.1)}$$

Donde λ_k es la tasa de fallo del nodo sensor k y t es el período temporal en el que se avalúa.

1.2.2. Escalabilidad

Los nuevos diseños deben ser capaces de trabajar con un número de nodos del orden de centenares, millares, e incluso, dependiendo de la aplicación, millones. También deben tener en cuenta la alta densidad, que puede llegar hasta algunos centenares de nodos sensores en una región, que puede ser menor de 10 metros de diámetro.
La densidad μ puede ser calculada de acuerdo con [4] como:

$$\mu(R) = (N \cdot \pi R^2) / A \qquad \textbf{(1.2)}$$

dónde *N* es el número de nodos sensores esparcidos en una región *A*, y *R* es el alcance de la transmisión radio. Básicamente, $\mu(R)$ nos da el número de nodos dentro del radio de transmisión de cada nodo en una región *A*.

1.2.3. Costes de producción

Dado que las redes de sensores consisten en un gran número de nodos sensores, el coste de un nodo individual es muy importante para justificar el coste completo de la red. Si el coste de la red es más caro que el despliegue de sensores tradicionales, la red sensora no está justificada desde el punto de vista económico.
El coste de un nodo sensor debería ser mucho menor a 1 dólar para permitir que la red de sensores fuera viable.
El coste de una interfaz radio Bluetooth, que es conocida por ser un dispositivo de bajo coste, es aún 10 veces más caro que el precio perseguido para un nodo sensor.

1.2.4. Limitaciones hardware

Un nodo sensor está constituido por cuatro componentes básicos, como muestra la figura 1.2: una unidad sensora, una unidad procesadora, una unidad transceptora, y una unidad de energía. Pueden tener también componentes adicionales dependiendo de su aplicación como un sistema de localización, un generador de energía o un movilizador.

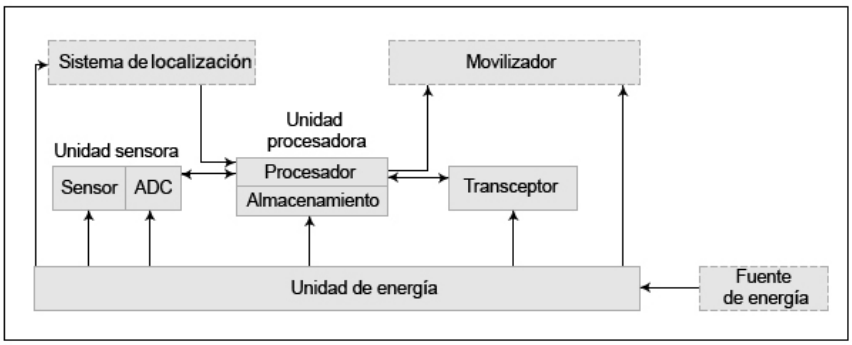


Fig.1.2 Estructura de un nodo sensor

Las señales analógicas producidas por los sensores basadas en el fenómeno observado son convertidas a señales digitales por el ADC, para ser pasadas después a la unidad procesadora.
La unidad de proceso, generalmente asociada a una pequeña unidad de almacenamiento, maneja los procedimientos necesarios para que el nodo

sensor colabore con los demás en la realización de las tareas de percepción asignadas.
Una unidad transceptora conecta el nodo a la red.
Uno de los componentes más importantes de un nodo sensor es la unidad de energía. Las unidades de energía pueden ser abastecidas por unidades de recogida de energía como células solares.
La mayoría de las técnicas de enrutamiento de las redes de sensores y las tareas de percepción requieren conocimiento de la localización con gran exactitud. Por ello, es habitual que un nodo sensor tenga un sistema de localización.
Todas estas subunidades pueden necesitar caber en un módulo del tamaño de una caja de cerillas. El tamaño requerido puede ser incluso menor que un centímetro cúbico.

1.2.5. Topología

El despliegue de un gran número de nodos densamente requiere un trato cuidadoso para permitir el mantenimiento de la topología.
Se pueden dividir las tareas de mantenimiento y cambio de la topología en tres fases:

- *Pre-despliegue y despliegue*: Los nodos sensores pueden ser arrojados en masa o colocados uno por uno en el campo sensor.

- *Post-despliegue*: Después del despliegue, los cambios de topología son debidos a: cambios en la posición de los nodos sensores, accesibilidad (debido a interferencias intencionadas (jamming), ruido, obstáculos móviles, etc), energía disponible, funcionamiento defectuoso y detalles de las tareas encomendadas.

- *Despliegue de nodos adicionales*: Nodos sensores adicionales pueden ser desplegados en cualquier momento para reemplazar nodos defectuosos o debido a cambios en la dinámica de las tareas.

1.2.6. Entorno

Los nodos sensores son desplegados densamente bien muy cerca o directamente en el interior del fenómeno a ser observado. Por consiguiente, normalmente trabajan desatendidos en áreas geográficas remotas. Pueden estar trabajando en el interior de maquinaria grande, en el fondo del océano, en un área contaminada biológicamente o químicamente, en un campo de batalla más allá de las líneas enemigas, y en edificios y hogares.

1.2.7. Medio de transmisión

En una red de sensores multisalto, los nodos comunicadores están conectados mediante un medio inalámbrico. Estas conexiones pueden estar formadas por medios radio, infrarrojo o óptico.
Mucho del hardware actual para redes de sensores está basado en RF. El nodo sensor inalámbrico AMPS descrito en [5] utiliza un transceptor a 2,4 Ghz compatible con Bluetooth. El dispositivo sensor de bajo consumo descrito en [6] utiliza un transceptor RF de un solo canal que opera a 916 Mhz.

Otro posible modo de comunicación entre nodos en redes de sensores es mediante infrarrojos. La comunicación por infrarrojos no necesita licencia y es robusta frente a interferencias producidas por dispositivos eléctricos. Los transceptores basados en infrarrojos son baratos y fáciles de construir.
Otro desarrollo interesante es el del Smart Dust [7], que es un sistema autónomo de percepción, computación y comunicación que utiliza el medio óptico para transmitir.
Ambos medios, infrarrojo y óptico requieren de visión directa entre el nodo o nodos transmisores y receptores.

1.2.8. Consumo energético

El nodo sensor inalámbrico siendo un dispositivo microelectrónico sólo puede estar equipado con una fuente energética limitada (< 0,5 Ah, 1.2 V).
En los escenarios de algunas aplicaciones, la recarga de los recursos energéticos puede ser imposible. El tiempo de vida de los nodos sensores, en consecuencia, muestra una gran dependencia del tiempo de vida de la batería. En una red sensora ad-hoc multisalto, cada nodo desempeña el doble rol de origen de información y enrutador de información.
El funcionamiento defectuoso de algunos nodos puede causar cambios de topología significantes y puede requerir re-enrutamiento de los paquetes y reorganización de la red. De aquí que, la conservación y administración energética tomen una importancia adicional.

1.3. Red sensora VS Red Ad-hoc

A pesar de que muchos protocolos y algoritmos provenientes de las redes inalámbricas ad-hoc tradicionales han sido propuestos, éstos no se acomodan bien a las características únicas y los requerimientos de las aplicaciones de las redes de sensores.
Para ilustrar el impacto de estas limitaciones, vamos a realizar una mirada más próxima a los esquemas MAC utilizados en otras redes inalámbricas y analizaremos porque no pueden ser adoptados en el escenario de una red de sensores.

En un sistema celular, las estaciones base forman una infraestructura (nervio) cableada. Un nodo móvil está solamente a un salto de la estación base más cercana. A este tipo de red se la conoce comúnmente en literatura como “basada en infraestructura”.
La meta principal del protocolo MAC en estos sistemas es la provisión de una alta calidad de servicio (QoS) y eficiencia del ancho de banda. La conservación de la energía asume un papel secundario dado que las estaciones base tienen suministro energético ilimitado y el usuario móvil puede recargar las exhaustas baterías. De aquí, el acceso al medio se inclina invariablemente hacia una estrategia dedicada a la asignación de recursos.
Tal esquema de acceso es impractible para redes de sensores dado que no hay un agente controlador central como la estación base. Esto convierte la sincronización completa de la red en una tarea difícil.
Además, la eficiencia energética influencia directamente el tiempo de vida de la red en una red de sensores y por consiguiente es de gran importancia.

Bluetooth y la red ad hoc móvil (MANET) son probablemente los más cercanos a las redes de sensores. La topología de Bluetooth es una red en estrella donde un nodo maestro puede llegar a tener hasta siete nodos esclavos conectados mediante wireless formando una piconet. Cada piconet utiliza un esquema central de asignación de acceso al medio mediante TDMA y un patrón de salto de frecuencias (frequency hopping).
La potencia de transmisión se sitúa típicamente alrededor de los 20 dBm y el rango de transmisión es del orden de decenas de metros.
El protocolo MAC de una MANET tiene la tarea de formar la infraestructura de la red y mantenerla frente a la movilidad. Por consiguiente, la primera meta es la provisión de una alta calidad de servicio (QoS) bajo condiciones móviles. A pesar de que los nodos son dispositivos portátiles alimentados por baterías, éstas pueden ser recargadas por el usuario, y en consecuencia el consumo energético tiene una importancia secundaria.
En contraposición con estos dos sistemas, la red de sensores puede tener un número mucho mayor de nodos. La potencia de transmisión (cercana a los 0 dBm) y el alcance radio de un nodo sensor son mucho menores que en Bluetooth o MANET. Los cambios de topología son más frecuentes en una red de sensores y pueden ser atribuidos tanto a movilidad de los nodos como a fallos. La tasa de movilidad en una red sensora se espera que sea mucho más lenta que en una MANET, los cambios de topología son en su mayoría debidos a fallos.
En esencia, la importancia primordial del ahorro energético para prolongar el tiempo de vida de la red en una red de sensores significa que ninguno de los protocolos MAC de Bluetooth o MANET existentes puede ser utilizado directamente.
Para ilustrar este punto, las diferencias entre las redes de sensores y las redes ad-hoc son:

- El número de nodos sensores en una red de sensores puede ser varios ordenes de magnitud más grande que los nodos de una red ad-hoc.
- Los nodos sensores están desplegados muy densamente.
- Los nodos sensores son propensos a fallar.
- La topología de una red de sensores varía con mucha frecuencia.
- Los nodos sensores utilizan principalmente comunicaciones broadcast, mientras que la mayoría de las redes ad-hoc están basadas en comunicaciones punto a punto.
- Los nodos sensores están limitados en potencia, capacidades de cómputo, y memoria.
- Los nodos sensores no disponen de una número de identificación global (ID) debido a la gran cantidad de overhead que esto supondría y al gran número de sensores desplegados.

Muchos investigadores están actualmente enfrascados en el desarrollo de diseños que cumplan estos requerimientos, algunos de estos diseños se van a mencionar y describir escuetamente en el apartado de estado del arte que viene a continuación.

CAPÍTULO 2: SISTEMAS DE POSICIONAMIENTO Y LOCALIZACIÓN

2.1. Fundamentos de la localización

Hasta ahora hemos conocido las redes de sensores, pero nuestro proyecto requiere del conocimiento de otro gran eje temático: los sistemas de localización. Debemos conocer cómo funcionan y cuáles son sus principios para ser capaces de diseñar uno que además se ajuste a redes de sensores.

2.1.1. Características de los sistemas de localización

2.1.1.1. La posición física y la localización simbólica

Un sistema de localización puede proporcionar dos clases de información: física y simbólica. GPS proporciona posiciones físicas. Por ejemplo, nuestro edificio está situado en 47,39º 17'' N – 122,18º 23'' O, a una elevación de 20,5 metros.
En contraste, la localización simbólica abarca ideas abstractas acerca de dónde se encuentra algo: en la cocina, junto a un buzón, en un tren que se aproxima a Castelldefels, etc.
Las aplicaciones pueden usar también la posición física para determinar un conjunto de información simbólica. Por ejemplo, una aplicación puede emplear una simple posición física GPS para encontrar la impresora más cercana.
Algunos ejemplos de tecnologías de localización simbólica son los escáneres de código de barras, y los sistemas que monitorizan la actividad de login en ordenadores.

2.1.1.2. Absoluto contra relativo

Un sistema de localización absoluto utiliza una rejilla de referencia compartida para todos los objetos localizados.
Por ejemplo, todos los receptores GPS utilizan latitud, longitud y altitud para señalar la localización. Dos receptores GPS situados en la misma ubicación señalarán lecturas de posición equivalentes, y 47,39º 17'' N – 122,18º 23'' O se refiere al mismo lugar independientemente del receptor GPS.
En un sistema relativo, cada objeto puede tener su propio marco de referencia.
Podemos utilizar la triangulación para determinar una posición absoluta a partir de múltiples lecturas relativas si conocemos la posición absoluta de los puntos de referencia.

2.1.1.3. Computo del cálculo de localización

Algunos de los sistemas que proporcionan capacidad de localización insisten en que el objeto que está localizado compute su propia posición.
Este modelo asegura la privacidad garantizando que ninguna otra entidad pueda saber dónde está el objeto a menos que éste, específicamente, tome la

decisión de divulgar esa información. Por ejemplo, los satélites GPS en órbita no tienen conocimientos sobre quién usa las señales que transmiten.
Por contra, otros sistemas requieren que el objeto ubicado transmita periódicamente, responda, o, por otra parte, emita telemetría, para permitir a la infraestructura externa que lo ubique.
Los sistemas de localización de Insignia Personal, los códigos de barras y la identificación de etiqueta de radiofrecuencia pertenecen a esta categoría.

2.1.1.4. La exactitud y la precisión

Un sistema de localización debe informar sobre localizaciones con exactitud y consistencia de medida a medida.
Algunos receptores GPS económicos pueden ubicar posiciones a menos de 10 metros para aproximadamente el 95% de las mediciones. Las unidades diferenciales más costosas alcanzan exactitudes de 1 a 3 metros en un 99 % de las veces. Estas distancias denotan la exactitud de la información de posición que GPS puede proporcionar. Los porcentajes denotan la precisión, o cuán a menudo podemos esperar conseguir esa exactitud.
A menudo, evaluamos la exactitud de un sistema de localización para determinar si es apropiado para una aplicación en particular.

2.1.1.5. La escala

Un sistema de localización puede ser capaz de ubicar objetos mundialmente, dentro de un área metropolitana, en todo un campus, en un edificio en particular, o dentro de una habitación. Además, el número de objetos que el sistema puede ubicar con cierta infraestructura o durante un tiempo podría estar limitado.
Para tasar la escala de un sistema de localización, consideramos su área de cobertura por unidad de infraestructura y el número de objetos que el sistema puede ubicar por unidad de infraestructura por intervalo de tiempo. El tiempo es una consideración importante debido al ancho de banda limitado disponible para detectar objetivos..

2.1.1.6. El reconocimiento

Para aplicaciones que tienen que reconocer o clasificar objetos ubicados para realizar una acción específica basada en su localización, es necesario un mecanismo de identificación automático.
Los sistemas con la capacidad de reconocimiento pueden reconocer solamente algunos tipos de características. Por ejemplo, las cámaras y los sistemas de visión pueden distinguir el color o la forma de un objeto fácilmente pero no pueden reconocer automáticamente a personas individuales.
Una técnica general para suministrar la capacidad de reconocimiento asigna nombres o identificaciones globalmente únicas (GUID) para los objetos que el sistema localiza. Una vez que una etiqueta, insignia, o una etiqueta sobre un objeto revela su GUID, la infraestructura puede acceder a una base de datos

externa para encontrar el nombre, el tipo, u otra información semántica sobre el objeto. La infraestructura también puede invertir el modelo de GUID para emitir documentos de identidad tales como URLs que los objetos móviles pueden reconocer y usar [9].

2.1.1.7. El coste

Podemos tasar el coste de un sistema localización de muchas maneras.
Los costes de tiempo incluyen factores como el tiempo de instalación y el tiempo necesario para la administración del sistema. Los costes de espacio involucran la cantidad de infraestructura a instalar y el tamaño del hardware.
Los costes de capital incluyen factores como el precio por unidad móvil o elemento de infraestructura y los salarios del personal de soporte.

2.1.1.8. Las limitaciones

Algunos sistemas no funcionarán en ciertos ambientes.
Una dificultad con el GPS es que los receptores generalmente no pueden detectar las transmisiones de los satélites en interiores.
En algunos casos, los sistemas colocados que usan la misma frecuencia de operación experimentan interferencia. En general, analizamos las limitaciones funcionales considerando las características de las tecnologías subyacentes que implementan el sistema de localización.

2.1.2. Técnicas de localización

La triangulación, el análisis de la escena, y la proximidad son las tres técnicas principales para la localización automática. Las técnicas de localización pueden emplearse individualmente o de forma combinada.

2.1.2.1. Triangulación

La técnica de triangulación utiliza las propiedades geométricas de los triángulos para computar localizaciones de objetos.
La triangulación se puede dividir en las subcategorías de lateración, usando medidas de distancia, y angulación, usando sobre todo medidas de ángulos u orientaciones.

Lateración
El término lateración significa para las mediciones de distancia, lo que angulación significa para los ángulos. La lateración computa la posición de un objeto midiendo su distancia de múltiples posiciones de referencia.
Calcular la posición de un objeto en dos dimensiones requiere las mediciones de distancia de 3 puntos no alineados. Para 3 dimensiones, se necesitan mediciones de distancia de 4 puntos no coplanarios.

Hay tres enfoques generales para medir las distancias requeridas por la técnica de lateración.

1. **Directo.** La medición directa de distancia usa una acción o movimiento físico. Por ejemplo, un robot puede extender una sonda hasta que toca algo sólido o toma mediciones con una cinta de medir. Las mediciones de distancias directas son simples de comprender pero difíciles de obtener automáticamente debido a las complejidades involucradas en coordinar movimientos físicos autónomos.

2. **Tiempo - de - vuelo.** Medir la distancia de un objeto hasta un punto P utilizando el tiempo-de-vuelo implica medir el tiempo que lleva el viajar entre el objeto y el punto P con velocidad conocida. El objeto, en sí mismo puede estar moviéndose, o como es más típico, el objeto es aproximadamente estacionario y estamos observando la diferencia de tiempo entre la emisión y la recepción de una señal transmitida. Este enfoque es el que utilizaremos para diseñar nuestro mecanismo de localización para redes de sensores.
Un impulso de ultrasonido transmitido por un objeto y llegando al punto P 14.5 milésimas de segundo después nos permite saber que el objeto está a 5 metros del punto P. Medir el tiempo-de-vuelo de luz o radio también es posible pero requiere relojes con una resolución mucho más elevada (de seis órdenes de magnitud).
Hacer caso omiso de los pulsos que llegan al punto P con una ruta indirecta (y por lo tanto más larga) causada por reflexiones en el ambiente es un desafío para medir el tiempo-de-vuelo ya que los pulsos directos y reflejados parecen idénticos.
Otro asunto a la hora de tomar mediciones de tiempo-de-vuelo es el acuerdo sobre la hora. Cuando solamente se necesita una medición, como un sonido de ida y vuelta (round-trip sound) o reflexiones de radar, el "acuerdo" es simple porque el objeto que transmite es también el receptor y sólo debe mantener su propio tiempo con suficiente precisión para computar la distancia. Sin embargo, en un sistema como GPS, el receptor no está sincroniza con los satélites transmisores y por tanto no puede medir con precisión el tiempo que tardó la señal a alcanzar el suelo desde el espacio.
Entre los sistemas de localización-detección del tiempo-de-vuelo se incluyen GPS, el Sistema de Localización Active Bat [10], el Cricket Location Support System [11] y la tecnología PulsON Time Modulated Ultra Wideband [12].

3. **Atenuación.** La intensidad de una señal emitida disminuye cuanto más se incrementa la distancia desde la fuente de emisión. El decrecimiento relativo a la intensidad original es la atenuación. Dada una función correlacionando la atenuación y la distancia para un tipo de emisión y fuerza original de emisión, es posible calcular la distancia desde un objeto a algún punto P midiendo la fuerza de la emisión cuando llega a P.
En ambientes con muchos obstáculos como un espacio interno de una oficina, medir la distancia usando la atenuación es generalmente menos exacto que el tiempo-de-vuelo. Las cuestiones que afectan a la propagación de señal, como el reflejo, refracción, y el multicamino, causan que la atenuación tenga una

correlación pobre con la distancia, resultando en cálculos inexactos e imprecisos de distancia.

El sistema SpotON ad hoc de Localización implementa la medición de la atenuación usando etiquetas de bajo coste [13]..

Angulación

La Angulación es similar a la lateración, excepto que, en lugar de las distancias, los ángulos son usados para determinar la posición de un objeto. En general la angulación bidimensional requiere dos mediciones de ángulo y una medición de longitud. En tres dimensiones, una medición de longitud, una de acimut, y dos de ángulo son necesarias para especificar una posición con precisión.
Una excelente aplicación de la tecnología de angulación son las matrices de antenas múltiples, que con una separación conocida miden el momento de la llegada de una señal.

2.1.2.2. Análisis de escena

La técnica de localización de análisis de escena usa características de una escena observada desde un punto de vista particular para obtener conclusiones sobre la localización del observador o de los objetos en la escena. Generalmente las escenas observadas se simplifican para obtener rasgos que sean sencillos de representar y comparar (por ejemplo, la forma de siluetas en el horizonte)
En el análisis de escenas estáticas, las características observadas son buscadas en un conjunto de datos predeterminados que las mapea a localizaciones de objeto. En contraste, el análisis de escenas diferenciales observa la diferencia entre sucesivas escenas para estimar la localización.
La ventaja del análisis de escena es que la posición de objetos puede ser inferida utilizando la observación pasiva y rasgos que no se corresponden a ángulos geométricos o distancias. Como hemos visto, la medición de magnitudes geométricas a menudo requiere movimiento o la emisión de señales, donde ambas pueden comprometer la privacidad y pueden requerir más energía. La desventaja del análisis de escena es que el observador necesita tener acceso a las características del entorno con el cual comparará sus escenas observadas.
Más aun, los cambios en el entorno de forma que se alteren los rasgos percibidos de las escenas pueden requerir la reconstrucción del conjunto de datos predefinidos, o la creación de un conjunto de datos completamente nuevo.
El sistema de localización RADAR, de Microsoft Research, es un ejemplo de lo último. RADAR utiliza un dataset de mediciones de intensidad de señal creado observando las transmisiones de radio de un dispositivo de red inalámbrica 802.11 en muchas posiciones y orientaciones dentro de un edificio [14]. La localización de otros dispositivos de red 802.11 puede ser calculada realizando una comparación de tabla sobre el dataset preconstruido.

2.1.2.3. Proximidad

Una técnica de localización de proximidad entraña el determinar cuándo un objeto está cerca de una posición conocida. La presencia del objeto es detectada utilizando un fenómeno físico con alcance limitado. Encontramos tres enfoques generales para detectar la proximidad:

1. Detectar el contacto físico**.** Detectar el contacto físico con un objeto es el tipo más básico de percepción de proximidad. Las tecnologías para percibir el contacto físico incluyen sensores de presión, sensores de tacto, y detectores de campo capacitivo. La detección del campo capacitivo se ha usado para implementar un Touch-Mouse [15] y Contact, un sistema para la comunicación intracorporal de datos entre objetos en contacto directo con la piel de una persona [16].

2. Monitorizar los puntos de acceso inalámbrico celular. Monitorizar cuándo un dispositivo móvil está en el alcance de uno o más puntos de acceso en una red inalámbrica celular es otra implementación de la técnica de localización de proximidad.
Los ejemplos de tales sistemas incluyen el Active Badge Location System (Sistema de localización de Insignia Activa) [17] y el sistema ParcTAB de Xerox [18], que utilizan ambos células infrarrojas difusas en un entorno de oficina, y el "Carnegie-Mellon Wireless Andrew" [19], que usa una red inalámbrica por radio tipo 802.11 para todo el campus.

3. Observando sistemas de identificación automáticos**.** Una tercera implementación de la técnica de detección de proximidad utiliza sistemas de identificación automáticos como en los terminales punto de venta de tarjetas de crédito y etiquetas de identificación como los sistemas electrónicos de pagado de peajes en autopistas.
Si el dispositivo que escanea la etiqueta, se comunica con la etiqueta, o monitoriza la transacción, tiene una localización conocida, la localización del objeto móvil puede ser inferida.

Los enfoques de proximidad pueden necesitar ser combinados con sistemas de identificación si ellos no incluyen un método para realizar identificaciones en la detección de proximidad. Las etiquetas de ganado poseen firmas únicas que identifican a los animales individuales. De forma similar ocurre para los teléfonos celulares. En contraste, el Touch Mouse [15] y los sensores de presión, requieren un sistema de identificación auxiliar.

2.2. Sistemas de posicionamiento global

Existen actualmente tres sistemas de posicionamiento global: GPS, GLONASS y el reciente GALILEO.
Los tres sistemas comparten un mismo principio de funcionamiento basado en la medida simultánea de la distancia entre el receptor y al menos 4 satélites.

Estas distancias se obtienen por medio del retardo temporal que existe desde el momento en que el satélite emite la señal hasta el momento en que el receptor la recibe.
De este modo, con la información proporcionada por cada satélite el receptor podrá obtener la ecuación de un esferoide, que resultará de todos aquellos puntos que se encuentren a una misma distancia del satélite (distancia que se habrá obtenido a partir del retardo temporal). La intersección entre todos los esferoides generados por las ecuaciones proporcionadas por los satélites nos definirá la posición del usuario, como se muestra en la figura 2.1.

Los satélites emiten dos portadoras a la misma frecuencia. Estas portadoras se encuentran moduladas en fase por diferentes códigos pseudoaleatorios.
El receptor calcula la correlación entre el código recibido y el código del satélite cuya señal pretende detectar, permitiéndole esto diferenciar entre las señales de los diferentes satélites en primera instancia, y calcular posteriormente el retardo temporal.

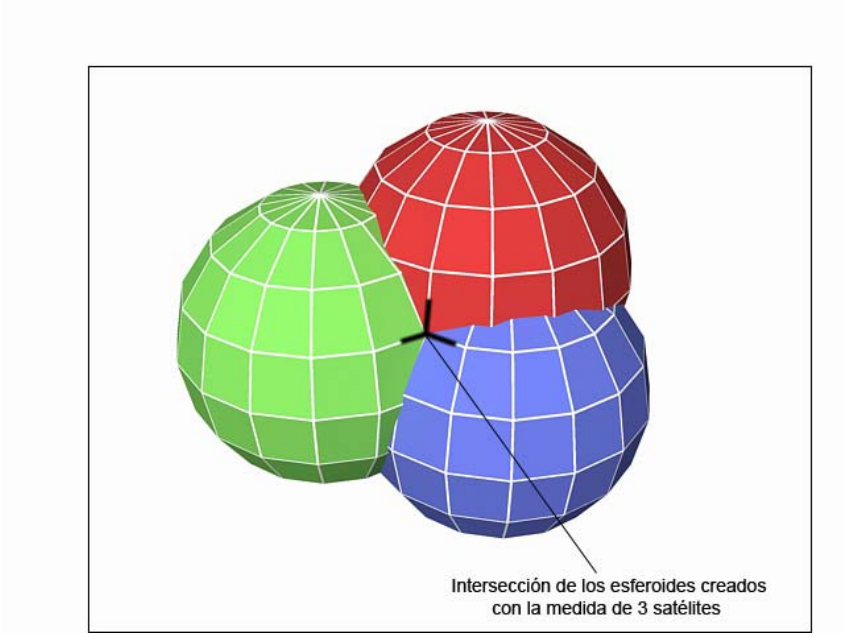


Fig.2.1 Intersección de esferoides para determinar la posición

Como puede deducirse del empleo de esta técnica, el receptor debe conocer de antemano los códigos pseudoaleatorios que utiliza cada uno de los satélites, sino no podría realizar el cálculo de la correlación.
A priori, por la definición que se ha dado, respaldada por la figura anterior, debería bastarnos con la información de 3 satélites para poder ubicar perfectamente al usuario receptor en las tres coordenadas del espacio tridimensional (X,Y,Z).
En realidad, esta apreciación es totalmente cierta, pero para ser posible requeriría de una gran precisión en la medida así como una gran estabilidad de los relojes. Este segundo requisito se cumple a la perfección en cuanto a los satélites se refiere, ya que éstos van equipados con osciladores atómicos de cesio o de rubidio, con precisión del orden de 10^{-12} segundos en el peor de los casos; sin embargo equipar un receptor terrestre con un oscilador de estas características sería económicamente inviable.

La solución para conseguir una medida aceptable, teniendo en cuenta este hecho, pasa por añadir una nueva incógnita al sistema que nos permita quantificar y tener en cuenta la deriva entre el reloj del sátelite y el reloj del equipo receptor.
La necesidad de una incógnita adicional requiere también que para la resolución del sistema de ecuaciones se disponga de una medida de más, es decir, de un 4º satélite (en caso contrario tendríamos un sistema de tres ecuaciones con cuatro incógnitas, sistema irresoluble).

El término pseudodistancia hace referencia a la medida que obtiene el receptor, y que en realidad es la distancia real más una distancia errónea (más o menos, el error puede ser positivo o negativo) provocada por la deriva entre sus relojes que he explicado anteriormente.
En caso de existir más de 4 satélites visibles, se consigue mejorar la precisión que será capaz de obtener el equipo receptor en cuanto a su posición, así como el tiempo de procesado, ya que a medida que tengamos más satélites obtenemos un sistema con más ecuaciones que incógnitas, circunstancia que simplifica su cálculo.

2.3. Estado del arte de sistemas de posicionamiento para redes de sensores

2.3.1. Tipos de esquemas de localización

2.3.1.1. Esquemas de localización basados en medidas de alcance

Estos esquemas encuentran la localización de un nodo de la red a partir del uso independiente o combinado de alguna de las siguientes medidas:

- TOA (Time of Arrival) : Tiempo de llegada
- TDOA (Time Difference on Arrival): Diferencia de tiempos de llegada
- AOA (Angle of Arrival): Ángulo de llegada
- RSSI (Received Signal Strength Indicador): Potencia de la señal recibida

Aunque se han propuesto esquemas de localización de este tipo para redes de sensores, la realidad es que su uso es poco conveniente por sus características intrínsecas: necesidad de un hardware caro y gran consumo energético.

2.3.1.2. Esquemas de localización no basados en medidas de alcance

Estos esquemas permiten que cada nodo de la red determine su posición basándose en el conocimiento de los saltos que lo separan de cada nodo de referencia.
Los nodos de referencia son nodos especiales que conocen su posición de antemano al inicio del sistema de localización interno de la red. Este

conocimiento puede deberse a que estén dotados de sistemas de posicionamiento especiales (por ejemplo, receptores GPS) o bien a que hayan sido desplegados de forma manual y se les haya indicado su posición.
En caso de ser necesarias operaciones computacionales de mayor complejidad y en consecuencia mayor gasto energético, éstas son realizadas por los nodos de referencia que posteriormente propagan el resultado al resto de los nodos.
En literatura, muchos de los grupos investigadores se han puesto de acuerdo en denominar a los nodos de referencia nodos ancla (Anchor Nodes).

2.3.2. GPS-less low-cost outdoor localization

La mayoría de los estudios posteriores sobre algoritmos de localización, se refieren a este diseño como "*Centroid scheme*", sin embargo, se ha preferido denominarlo con el nombre original con que fue publicado en [21]

Este diseño se basa en el emplazamiento de múltiples nodos de referencia con áreas de cobertura solapadas. Estos nodos de referencia están situados en posiciones conocidas formando una rejilla para que sus solapamientos sean regulares y transmiten de forma periódica *beacons* que contienen información sobre su posición.
Cada nodo móvil escucha durante un periodo de tiempo *t* y recoge todos los *beacons* que recibe de diferentes nodos de referencia. Cuando termina este periodo de escucha, el nodo evalúa basándose en un parámetro llamada métrica de conectividad que nodos de referencia tiene más cerca.
Una vez hecha esta selección, el nodo calcula su posición ubicándose en el centro de la región de intersección de las áreas de cobertura de los nodos de referencia próximos.
Es por este hecho por el que se le denomina "*Centroid scheme*".
Aumentando la densidad de los nodos de referencia se reduce la granularidad de las áreas de localización (áreas de solapamiento de nodos de referencia vecinos) y en consecuencia, mejora la exactitud de la posición estimada.

2.3.3. DV based positioning

Este esquema de localización se propone en [22] y se basa en la aplicación de un mecanismo muy similar al enrutamiento basado en vector de distancia.
Cada nodo de referencia inunda la red con un *beacon* que contiene su localización y un contador del número de saltos inicializado a uno. A medida que se retransmiten los *beacons*, el valor del contador se incrementa en cada nuevo salto.
Debido a que los nodos de la red retransmiten el *beacon*, un mismo nodo recibe varios *beacons* con información de un mismo nodo de referencia. De entre todos ellos, guarda el valor del contador que indique el número de saltos mínimo y ignora aquellos *beacons* con valores de contador mayores.
A través de este mecanismo, todos los nodos de la red (incluidos los demás nodos de referencia) consiguen información de la menor distancia, en número de saltos, que los separa de cada nodo de referencia.

La conversión entre número de saltos y distancia física se realiza estimando la distancia media por salto. Esta operación la realizan los nodos de referencia.
Una vez calculada la distancia por salto, el nodo de referencia propaga su valor a los nodos colindantes.
Cuando un nodo puede calcular una distancia estimada hasta 3 nodos de referencia, utiliza triangulación para calcular su posición.

2.3.4. Amorphous localization

El algoritmo *Amorphous localization* que se propone en [23] calcula la posición estimada de forma muy similar al algoritmo DV.
Primero, de igual forma que DV, cada nodo encuentra la distancia en saltos a cada nodo de referencia mediante la propagación de *beacons*.
De este modo la resolución de distancia es bastante pobre, un nodo sólo podrá estar situado a distancias múltiplo de la distancia media por salto.
Para mejorar esta baja resolución, una vez que un nodo conoce la distancia en saltos a los nodos de referencia, pregunta a sus nodos vecinos que distancia en saltos tienen ellos respecto a cada nodo de referencia y realiza un promedio de estos valores.
Sin embargo, toma una aproximación diferente para el cálculo de la distancia media por salto. Este algoritmo asume que la densidad de la red, n_{local} , se conoce a priori.
Finalmente, después de que cada nodo obtenga las distancias estimadas a tres nodos de referencia, calcula su posición mediante triangulación.
Utilizando únicamente tres nodos de referencia para la triangulación, se necesita una vecindad media de 15 nodos para asegurar una buena exactitud de la posición. Con el incremento de los nodos de referencia utilizados para realizar el cálculo de triangulación, puede reducirse significativamente el número de vecinos necesarios para conseguir una buena resolución.

2.3.5. APIT localization

El mecanismo de localización APIT, descrito en [20] requiere de una red de sensores heterogénea, donde un pequeño porcentaje de estos dispositivos debe estar equipado con transmisores de alta potencia y información de su localización obtenida mediante GPS o otros métodos.
Estos nodos, que llamaremos nuevamente nodos de referencia, envían periódicamente *beacons*. Para minimizar el área donde puede encontrarse un determinado nodo se utiliza un mecanismo llamado PIT (Point-in-Triangulation Test) . En este mecanismo, el nodo escoge 3 nodos de referencia de entre todos los que escucha (de los que recibe *beacons*) y prueba si se encuentra dentro del triángulo formado de interconectar estos tres nodos. APIT repite el test PIT con diferentes combinaciones de nodos de referencia hasta que las posibles combinaciones se terminan o hasta que consigue la exactitud requerida.
En este punto, APIT calcula el centro de gravedad (COG) de la intersección de todos los triángulos donde se ha demostrado que reside el nodo para determinar su posición estimada.

CAPÍTULO 3: ENTORNO DE TRABAJO

En este capítulo se va a realizar una descripción de los componentes, tanto hardware como software, que han sido necesarios para llevar a cabo el presente proyecto.

En el laboratorio se ha trabajado con un equipo Windows XP y un kit de desarrollo comercial de Crossbow.
En cuanto al software, se ha utilizado el sistema operativo TinyOS, especialmente diseñado para las características intrínsecas de los nodos sensores y sus aplicaciones de red potenciales, y en él programaremos aplicaciones para los nodos sensores en el lenguaje de programación nesC.
Dado que TinyOS requiere para su correcto funcionamiento del uso de diversos comandos y funcionalidades Linux, ha sido necesaria la instalación de la herramienta Cygwin, encargada de emular un entorno Linux en Windows.

Para nuestro proyecto desarrollaremos también una aplicación de pc para la que utilizaremos el lenguaje de programación Java, la edición estándar 1.4.2 del kit de desarrollo de software y el entorno de desarrollo integrado Jcreator V3 LE.

3.1. TinyOS

TinyOS es un sistema operativo de código libre basado en eventos destinado para utilizarse en sensores.
Después de introducirse en el mundo de las redes de sensores con el primer capítulo de este trabajo, es fácil darse cuenta de la utilidad y necesidad de un sistema operativo adaptado a sensores y basado en eventos.
La actividad de un nodo sensor está basada en los impulsos externos que recibe: mensajes radio de otros nodos sensores, detección de sonido, detección de una temperatura más alta de lo habitual y que debe generar un mensaje de alerta que se propague por la red, etc.
TinyOS utiliza un modelo de programación basado en el concepto de "*wiring*" (enlazar o cablear) componentes software para producir un programa final.
Este modelo de programación además, pone requisitos sobre como deben ser escritos los programas. TinyOS, siguiendo los patrones de las redes de sensores, toma en cuenta que puede haber muy pocos recursos disponibles (por ejemplo, 512 bytes de RAM) y que esto requiere una utilización de los recursos muy eficiente. Otro requerimiento tiene que ver con el concepto *wiring*, los programas deben ser capaces de mapear una sola llamada a función (input wire) para que sean llamadas múltiples funciones (output wires).

Además, TinyOS utiliza frecuentemente macros del preprocesador de C para permitir modos de compilación alternativos (como pueden ser simuladores).
La página en Internet donde se puede descargar el sistema operativo de distribución libre TinyOS es www.tinyos.net , además en esa página se encuentra la comunidad de desarrolladores y proporciona amplia documentación sobre el proyecto, así como listas de desarrolladores a las que uno se puede apuntar.

3.1.1. Modelo de componentes

En TinyOS la distinción entre el sistema operativo en sí y las aplicaciones es meramente semántica; ambas cosas se compilan de forma conjunta y comparten un mismo espacio de memoria.
A medida que los componentes de aplicación son probados, se vuelven más estables y crecen para ser componentes que utilicen otras aplicaciones, pueden acabar convirtiéndose en componentes del sistema.
Los componentes de TinyOS pueden dibujarse formando una pirámide invertida: encima de todo se encuentran los componentes del nivel de aplicación, mientras que la capa más baja la ocupan los componentes que se asientan directamente sobre el hardware.

3.1.1.1. Eventos, comandos y tareas

TinyOS presenta tres abstracciones de computación: los eventos, los comandos y las tareas.

Los comandos representan "llamadas hacia abajo" en la pirámide de componentes; es decir, un componente llama a comandos que pertenecen a componentes que se encuentran por debajo de él en la pirámide.
Los eventos representan "llamadas hacia arriba" en la pirámide de componentes; es decir, un componente advierte que ha sucedido un evento a componentes que se encuentran por encima de él en la pirámide.
La forma más precisa de llamar a los eventos sería capturadores de eventos, pues como ya indicábamos anteriormente un evento propiamente dicho es un suceso externo que tiene repercusión en la actividad de un nodo sensor.
Es importante diferenciar que los capturadores de eventos pueden llamar a comandos, pero un comando nunca puede señalizar un evento.

Las tareas son un mecanismo para computación asíncrona de larga duración. Una tarea se ejecuta sincronizadamente respecto a otras tareas, todas tienen la misma prioridad y se ejecutan en orden. Sin embargo, una tarea puede verse obligada a abandonar el procesador si llega un evento de mayor prioridad. Por lo tanto si realizamos tareas con requisitos de tiempo real, éstas deben ser cortas para evitar que puedan ser interrumpidas por sucesivos eventos.

También es crítico que el código llamado por un capturador de eventos sea lo más corto posible. Esto se debe a que normalmente se ejecutan como resultado de una interrupción, por tanto es posible que no puedan ser interrumpidos por otras interrupciones posteriores hasta su finalización, manteniendo detenida la ejecución del código principal y manteniendo en espera interrupciones que podrían ser críticas para la aplicación.

3.2. NesC

NesC es un nuevo lenguaje de programación, de sintaxis similar a C, en que está programado el sistema operativo TinyOS, así como sus librerías y sus aplicaciones.
Las siglas con que se denomina significan network embedded systems C.
Originalmente TinyOS estaba programado en C (hasta su versión 0.6), sin embargo, en la transición entre la versión 0.6 y la 1.0 se reimplementó todo el sistema operativo en nesC.
Las ventajas que proporcionaba nesC eran: aparición de las interfaces (de las que se hablará posteriormente), detección de errores de "*wiring*", generación automática de documentos (similar a javadoc(1)), y mayor facilidad para la optimización del código significativo del sistema operativo respecto al anterior modelo de TinyOS. Su mayor éxito fue conseguir que el código de TinyOS fuese más limpio, más sencillo de entender y más sencillo de escribir.

TinyOS define un número importante de conceptos que tienen su correspondencia en el lenguaje de programación nesC.
En primer lugar, las aplicaciones nesC están construidas por componentes con interfaces bidireccionales y bien definidas. En segundo lugar, nesC define un modelo concurrente, basado en tareas y capturadotes de eventos y capaz de detectar *data races* en tiempo de compilación.

3.2.1. Componentes

3.2.1.1. Especificación de los componentes

Una aplicación nesC consiste en uno o más componentes enlazados entre ellos para formar un ejecutable. Un componente proporciona y utiliza interfaces. Estas interfaces representan el único punto de acceso a un componente y son bidireccionales. Una interfaz declara un conjunto de funciones llamadas comandos que el proveedor de la interfaz debe implementar, y otro conjunto de funciones llamadas eventos que, en este caso, deben ser implementadas por el usuario de la interfaz.
Un componente puede utilizar o proporcionar múltiples interfaces y múltiples instancias de una misma interfaz (por ejemplo, un componente que necesita dos temporizadores deberá instanciar dos veces a la interfaz que proporcione el temporizador).

3.2.1.2. Implementación de los componentes

Existen dos tipos de componentes en nesC: módulos y configuraciones.
Los módulos proporcionan el código de la aplicación, implementando una o varias interfaces.

---------- (1) Javadoc es una herramienta de java que genera documentación básica para el programador a partir del código fuente. Mediante la extracción de los comentarios del propio código genera un juego de documentación en html. ----------

Las configuraciones se utilizan para ensamblar otros componentes conjuntamente, conectando las interfaces utilizadas por los componentes con las interfaces proporcionadas por otros. A esto se le llama "*wiring*", un concepto que de forma abstracta ya se introdujo en el apartado sobre TinyOS.
Cada aplicación nesC requiere de una configuración de alto nivel que enlace entre ellos los componentes utilizados.
NesC utiliza la misma extensión de archivo, .nc, para todos sus archivos, es decir, tanto para interfaces como para configuraciones y módulos. Sin embargo se proponen unas convenciones de nombre para diferenciar unos componentes de otros.

3.2.2. Modelo concurrente

TinyOS ejecuta únicamente un programa consistente en los componentes de sistema y los componentes propios necesarios para que funcione la aplicación.
Existen dos hilos de ejecución: las tareas y los capturadores de eventos.
Si recordamos, los capturadores de eventos lanzaban la ejecución de un código vinculado al evento ocurrido y/o llamaban a comandos que se encargaban así mismo de tareas específicas vinculadas al evento.
El lenguaje nesC debe tener conocimiento de que comandos y eventos serán ejecutados como parte de un capturador de eventos, para ello en su definición se les debe declarar con la palabra clave *async*.

Debido a que tareas y capturadores de eventos pueden ser detenidos para permitir la ejecución de código asíncrono de mayor prioridad, los programas en nesC son susceptibles a padecer *data races*(2). Las condiciones de aparición de *data races* pueden evitarse mediante el acceso a datos compartidos exclusivamente en el interior de las tareas o bien encapsulando todos los accesos dentro de secciones *atomic*.

Las secciones *atomic* aseguran la ejecución de todo el código que encapsulan, si llega una interrupción, ésta esperará hasta que la ejecución abandone la sección *atomic* para ejecutarse. El compilador de nesC advierte de *data races* potenciales en tiempo de compilación para que el programador pueda rediseñar su código.

3.3. Kit Crossbow

El kit de desarrollo del que se ha dispuesto en el laboratorio es de la marca comercial Crossbow, una de las más extendidas y conocidas en cuanto al desarrollo de kits de hardware para la investigación en redes de sensores inalámbricas.

---------- (2) Son situaciones que ocurren cuando diferentes procesos acceden a datos compartidos sin estar sincronizados. Pueden provocar compartamientos inesperados del programa por lo que es importante detectarlas a tiempo. ----------

Los componentes básicos son:

- Una placa de desarrollo modelo MIB510
- Plataformas de procesador y radio modelos mica2 y mica2dot
- Placas sensoras y de adquisición de datos modelos MTS310 y MTS510

Fig 3.1 *(A)* En la parte superior vemos el modelo mica2 (el rectangular) y el modelo mica2dot (el circular), y en la inferior la placa de desarrollo MIB510
(B) y (C) Diferentes modelos de placas sensoras

3.3.1. Placa de desarrollo MIB510

La placa de desarrollo MIB510 se interconecta con el ordenador mediante puerto serie y es la encargada de proporcionar el enlace necesario para que las aplicaciones escritas en el ordenador sean programadas en las plataformas mica2 y mica2dot.
Para programar las plataformas mica2, éstas deben conectarse mediante el conector Hirose de 51 pins a la placa de desarrollo, y siempre deben programarse con el interruptor en la posición de apagado. Si se conecta por error con el interruptor encendido corremos el peligro de que el mica2 padezca sobretensión.

Para programar las plataformas mica2dot, éstas deben conectarse por debajo mediante el conector circular de 19 pins que podemos observar en la figura 3.1. Las plataformas mica2dot no tienen interruptor de encendido y apagado, en cuyo caso lo que debemos evitar es conectarlas a la MIB510 con la pila de botón puesta (ya que con la pila puesta están permanentemente encendidas).

3.3.2. Plataformas de procesador y radio

Estas plataformas son comúnmente llamadas *motes*. Incluso la misma compañía Crossbow las denomina así en las hojas de especificaciones y la documentación que se pueden obtener en su página en internet www.xbow.com.

Estas plataformas contienen una memoria flash donde son descargadas las aplicaciones vía la placa de desarrollo.
Los rasgos más característicos del mica2 son la presencia de tres leds de colores rojo, verde y amarillo y la antena para transmitir y recibir señales radio.
En el caso de la plataforma mica2dot sólo consta de un led de color rojo y la antena, a diferencia del mica2, está soldada a la placa.

3.3.3. Placas sensoras y de adquisición de datos

Estas placas se añaden a los *motes* mediante los mismos conectores con que éstos se unen a la placa de desarrollo para ser programados.
La principal función de las placas es añadirles funcionalidades sensoras a las plataformas para acercar más las funcionalidades potenciales del escenario a las características de una red de sensores.
El sistema operativo TinyOS cuenta con componentes capaces de interactuar con estos sensores, de modo que puedan conseguirse aplicaciones donde la plataforma dotada de la placa sensora recoja ciertas medidas del entorno para luego transmitirlas en un mensaje radio al resto de los nodos de la red o al ordenador.

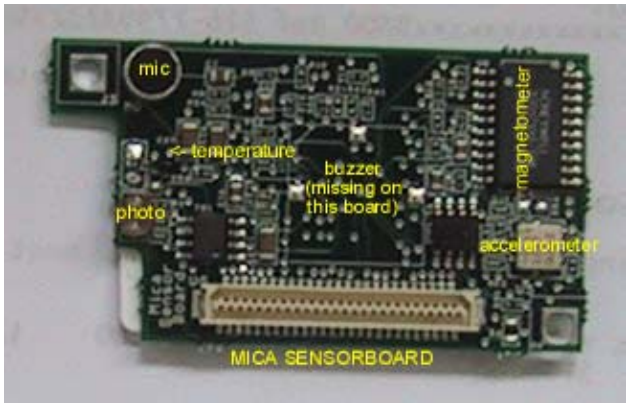


Fig 3.2. Placa sensora MTS310 con información de los sensores disponibles

CAPÍTULO 4: DESARROLLO DEL PROYECTO

4.1. Idea general

El objetivo de nuestro proyecto era realizar en primer lugar un estudio para cerciorarnos de si era viable acometer la tarea de implementar un mecanismo de localización en redes de sensores basado en sonido y en la técnica de triangulación.
Nuestra aproximación consiste en el cálculo del tiempo de vuelo del sonido entre nodos con posición conocida y un nodo con necesidad de ubicarse dentro de la red.
Prácticamente desde el principio se descartó la viabilidad de realizar el mecanismo de localización basado en tiempo de vuelo de la señal radio debido a los altos requisitos necesarios. Si recordamos, como se vio en el capítulo 2, pretender cálculos de distancia basados en señal radio supondría el uso de temporizadores de resolución tremendamente elevada.
A esto se le añade nuestra intención de realizar un mecanismo de localización con precisión de centímetros y encarado a discernir distancias en una red de nodos sensores densamente poblada.
Esta viabilidad dependía de muchos factores que serán explicados detalladamente en la siguiente sección de este capítulo.
A grandes rasgos, nuestras preocupaciones principales eran: capacidad de enviar un mensaje de inundación únicamente y exclusivamente a los nodos vecinos, existencia de un temporizador con resolución suficiente para conseguir a priori precisión de centímetros y correcto funcionamiento de la interacción entre los diferentes sensores que debían ser la base de nuestro sistema: altavoz y micrófono.

Una vez realizado el estudio de viabilidad con un resultado suficientemente satisfactorio, se pasó a la implementación de los códigos escritos en nesC de las dos aplicaciones que se requerían: una aplicación generadora de tonos, que ejecutaría el nodo ilocalizado, y una aplicación receptora de tonos y capaz de realizar cálculo de distancias basado en sonido que ejecutarían los nodos de referencia de nuestro escenario.
Debido a las restricciones energéticas y de cómputo intrínsecas a las redes de sensores, se optó posteriormente por un acercamiento al problema en que los nodos de referencia únicamente capturaban el tiempo de vuelo y esta información se transmitía al ordenador para que fuese éste quien realizase los cálculos de distancia y la triangulación del nodo ilocalizado.

Como uno de nuestros objetivos era conseguir que esta información no solo quedase en conocimiento de la red sino que pudiese ser monitorizada desde un ordenador, se implementó una aplicación para mostrar estos datos al usuario final de forma sencilla y agradable.

Nuestro escenario funciona de la siguiente manera:

- Tenemos un nodo en una posición desconocida (esta situación puede deberse a que sea un nodo nuevo que acaba de ser desplegado, por ejemplo), este nodo emite un mensaje radio de "sincronización" e inmediatamente después genera un tono de 4,3 Khz durante 1 segundo.

- El resto de nodos de la red se supone que ya conocen su posición y que el elemento de control (PC en nuestro caso) también las conoce. Estos nodos cuando reciben el mensaje radio de "sincronización" toman el valor de un temporizador en ese instante, y cuando detectan correctamente el tono de 4,3 Khz, toman otro valor. Con la resta de estos dos valores, podemos encontrar la distancia que separa el nodo receptor del nodo origen.

- Con suficientes medidas de distancia (3 nodos receptores) enviadas al elemento de control (PC), éste puede realizar triangulación para ubicar la posición en que se encuentra el nuevo nodo.

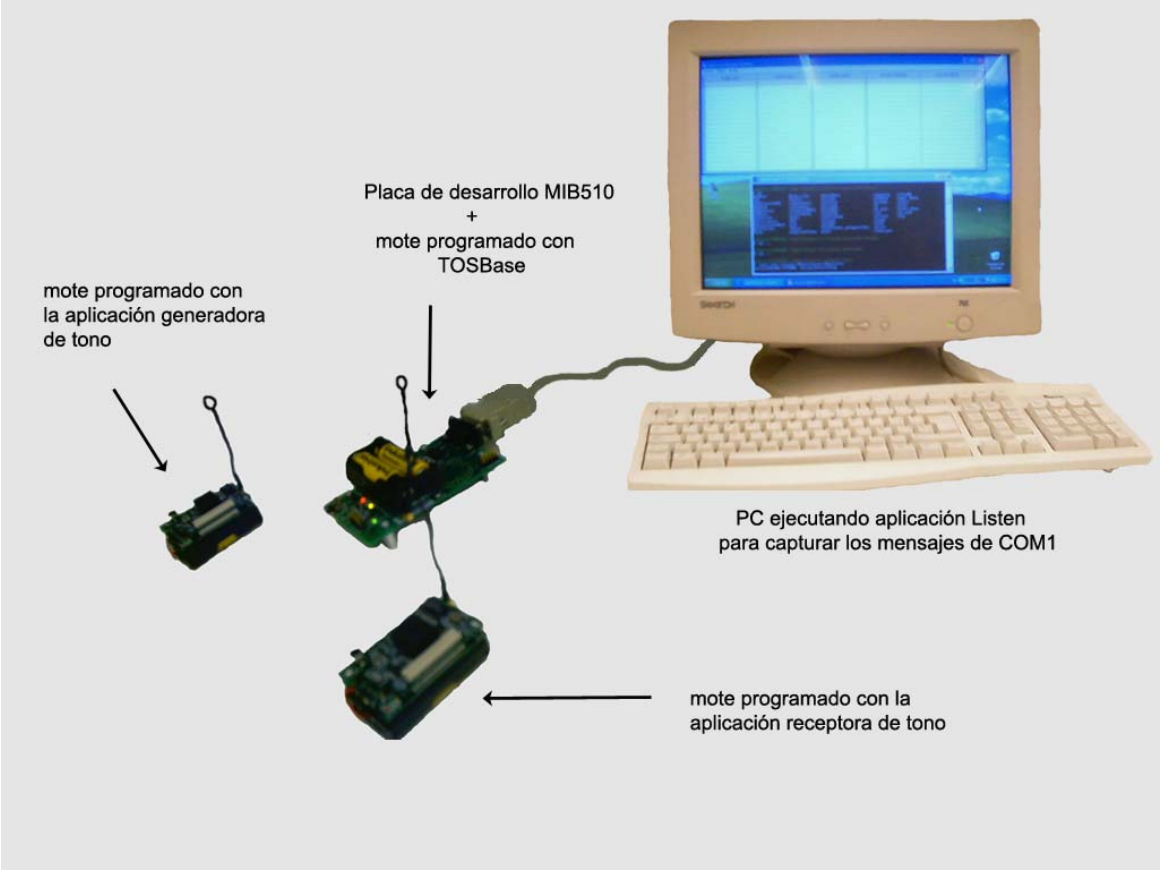


Fig 4.1. Escenario de pruebas

4.2. Estudio de viabilidad

4.2.1. Temporizadores

4.2.1.1. Timer

La primera solución que se estudió fue el uso del temporizador software que proporcionaba la interfaz de alto nivel *Timer.*
Esta interfaz era implementada por el componente del sistema *TimerC*, un componente ampliamente utilizado en los códigos de ejemplo que TinyOS proporciona para que un programador novel se familiarice con el funcionamiento, la semántica y los conceptos que entraña la programación en nesC.

Su principal e importante desventaja es que el temporizador tenía que ser inicializado en el código de la aplicación, hecho que generaba cierta incertidumbre sobre su momento de inicio (otras tareas y/o eventos podían interrumpir su inicialización) y posteriormente su precisión iba a verse muy degradada e iba a ser nuevamente motivo de incertidumbre por las interrupciones debidas a eventos que podían detener momentáneamente su ejecución.
Para intentar evitar estos sucesos indeseados se estudió la inclusión de las sentencias del temporizador en cláusulas *atomic* (cuya función, como se vio anteriormente, es definir un trozo de código de forma que éste se ejecute sin que ningún evento ni interrupción pueda interrumpirlo, y cuyo uso se extiende a los fragmentos del código más críticos como son los accesos a memoria o su escritura), pero esta solución tampoco era viable debido a que precisamente la recepción del tono es notificada a la aplicación mediante una interrupción.

Otra desventaja que se consideró relevante fue el hecho de que la interfaz *Timer* no proporcionaba ningún comando de inicialización infinita. En el momento de inicializar el temporizador debía especificarse cual sería su tiempo de vida.
Debido a que a priori era imposible conocer el alcance del tono sonoro, no se podía conocer cuanto tiempo era necesario tener al temporizador activo para que se pudiese capturar su valor en el momento en que fuese necesario.
A esto hay que añadirle que la interfaz no proporcionaba directamente comandos de consulta del valor del temporizador ni parecía contemplar la interrupción del temporizador cuando llegase un evento.

Por último, la resolución que nos proporcionaba este temporizador era de microsegundos.

4.2.1.2. Clock

Un estudio en mayor profundidad de las diferentes interfaces y componentes del sistema, nos permitió encontrar la interfaz *Clock*, que proporcionaba comandos de acceso a un reloj de la plataforma hardware.

Esta solución se presentaba más acorde a nuestras necesidades debido a que al ser un reloj hardware, la aplicación no debía preocuparse de que su funcionamiento fuese ininterrumpido, éste era así por definición.
Además, esta interfaz sí proporcionaba un comando para capturar el valor del reloj en el momento en que se desease, que era implementado por el componente *ClockC*.

Este reloj proporcionaba en su inicialización la capacidad de definirle un rango de trabajo determinado, es decir, una resolución determinada.
Esta resolución variaba desde los 32 ticks de reloj por segundo hasta los 32.768 ticks por segundo. Esta última cota de resolución, como se puede observar, nos proporcionaba una resolución más alta que microsegundos (serían 1000 ticks por segundo).

Esta solución fue aceptada e incluso se empezó la fase de programación de la aplicación sin percatarse de un problema muy grave que entrañaba su uso.
El comando encargado de capturar su valor, por definición, devolvía éste en una variable entera de 1 byte. Esto conllevaba serios problemas de truncamiento debido a que toda medida de distancia en ticks de reloj superaba con creces el valor capaz de almacenarse en un byte.

4.2.1.3. Systime

La interfaz *Systime* nos proporciona acceso a un reloj de la plataforma hardware y nos proporciona comandos para capturar su valor. Aquí acaba todo parecido con su antecesor en nuestra línea de investigación *Clock*.

En este caso, tenemos un reloj de frecuencia 921,6 Khz y una función que nos permite coger el valor de este contador mediante un entero de 32 bits (4 bytes). Este contador lo que nos va a devolver son los "ticks" de reloj.

La frecuencia lo que nos indica son los ticks por segundo, por tanto utilizando este reloj, cada segundo se contabilizarán 921600 ticks (que viene a darnos una resolución de microsegundos, 1/921600 ≈ 1exp-6).
En 32 bits podemos almacenar 2^{32} valores, por tanto, 4.294.967.296 ticks de reloj.

Por tanto, el tiempo máximo que puede separar a dos motes para que no "de la vuelta" el reloj es:

$$4.294.967.296 / 921.600 \approx 4660 \text{ segundos.}$$

Ocurrirá antes que el micrófono de un mote no tenga suficiente sensibilidad para apreciar el tono de 4,3 Khz (por atenuación ligada a la distancia) que no que se de la vuelta el contador (para tener los motes a distancia temporal a velocidad del sonido de 4660 segundos, deberían estar separados espacialmente más de 1500 km).
Por tanto este contador cumple todas nuestras expectativas.

4.2.2. Propiedades de inundación y retransmisión

4.2.2.1 Inundación

Estudiando el formato de los mensajes radio predefinidos por el sistema operativo TinyOS, puede comprobarse como, a menos que se indique lo contrario, todos los mensajes se transmiten con el campo de dirección destino indicando mensaje de inundación o mensaje broadcast.
La comprobación práctica se realizó con el código de ejemplo *SimpleCmd* y la aplicación de ordenador *BcastInject*.

Se programaron dos mica2 con la aplicación *SimpleCmd*, en que esperan la recepción de un comando por radio y lo ejecutan sin habilitar la posibilidad de retransmisión broadcast que proporciona la misma aplicación y que consiste en que cada nodo cuando recibe el paquete con el comando y lo ejecuta, lo reenvía de nuevo por RF.
Se programó otro mica2 como *Base* y se dejó conectado a la placa de desarrollo y posteriormente, mediante la aplicación *BcastInject* se introdujeron en la red sucesivos mensajes radio conteniendo los comandos red_on y red_off (estos comandos, como indican sus nombres, sirven para encender y apagar el led rojo del mica2, respectivamente).
Se comprobó que el mensaje con el comando era recibido y ejecutado correctamente por los dos mica2 programados con *SimpleCmd*.

También se realizó una segunda prueba que no requiriese de la intervención de un código ajeno en java (lenguaje de la aplicación *BcastInject*) ni de la necesidad del ordenador.
Se programaron dos mica2 con la aplicación *RfmToLeds* y uno con la aplicación *CntToRfm*. La aplicación *RfmToLeds* se encarga de visualizar el campo de datos que contienen los mensajes radio a través de los leds de la plataforma. La aplicación *CntToRfm* envía mensajes radio que contienen los valores sucesivos de un contador.
El resultado que se esperaba para comprobar el broadcast es que el valor del contador que el mica2 programado con *CntToRfm* enviara por radio fuese recibido y mostrado en los leds por los dos mica2 programados con *RfmToLeds.*
Siendo el resultado satisfactorio, se comprobó el envío de paquetes broadcast no desde un nodo Base sujeto a la placa de desarrollo, sino desde un nodo cualquiera de la red.

4.2.2.2. Retransmisión

Esta consideración cobra relevancia si se conocen los fundamentos en que se basa la implementación de nuestro mecanismo de localización.
Es por ello que sin entrar en detalles de implementación, que se verán más adelante, se va a describir el diseño del mecanismo.

Se pretendía descubrir si los nodos a más de un salto del nodo origen iban a recibir y en consecuencia procesar el mensaje radio de sincronismo.

Sin embargo, para que la aplicación funcione correctamente y los valores de distancia sean correctos, el mensaje radio que genera nuestro nodo origen sólo debe ser procesado por sus nodos vecinos, es decir, aquellos que se encuentran a un solo salto de él.

Si estos nodos vecinos realizasen retransmisión del mensaje radio del nodo origen a otros nodos (inundación de la red) y éstos nodos situados a N-saltos ($N \neq 1$) fuesen capaces de percibir el tono de 4,3 Khz, obtendríamos medidas falsas de distancia.
Esta medida falsa se daría por la situación siguiente:

- Estos nodos a N-saltos reciben el mensaje radio del nodo origen mediante reenvío, no directamente desde el nodo origen, y en consecuencia este mensaje les llega con un cierto retardo adicional (de proceso y reenvío en cada nodo intermedio). Por tanto la diferencia temporal entre la recepción del mensaje radio y la recepción del tono no va a ser correcta.

Un estudio de la interfaz genérica de comunicación (interfaz *GenericComm*) ha permitido comprobar que ésta no dota a los nodos de una capacidad inherente de reenvío de mensajes. Éstos únicamente serán reenviados si nosotros lo especificamos mediante código programado en la aplicación desarrollada.

Un ejemplo de esto podemos encontrarlo en la aplicación de ejemplo *SimpleCmd*.
Esta aplicación permite dos modos de funcionamiento, en uno de ellos sólo reciben y procesan el comando inyectado desde el PC los nodos que están a un salto, el otro, donde se utiliza un componente denominado *Bcast* permite que el comando inyectado se propague por la red indefinidamente (realmente el componente *Bcast* evita que un mismo nodo ejecute dos veces el comando fijándose en el número de secuencia del mensaje para decidir si es un mensaje nuevo o bien ya ha sido procesado anteriormente).
Para proporcionar a la aplicación la capacidad de reenvío de mensajes, ésta ha tenido que ser programada.

Sin embargo, no vamos a quedarnos en la comprobación teórica y vamos a realizar una comprobación a la práctica. Podría ser que los nodos tuviesen un código fijo por defecto además del que programamos nosotros que proporcionase esa capacidad.
Para realizar la comprobación práctica nos basta con nuestro escenario de ensayo. Tenemos tres nodos, el nodo origen programado con la aplicación generadora de tono, al que llamaremos a partir de ahora nodo1, un nodo receptor programado con la aplicación receptora de tono, desde ahora nodo2, y un tercer nodo utilizado como *Base* (este nodo debe estar programado con la aplicación TOSBase) que está colocado en la placa de desarrollo y que se encarga de transmitir los mensajes radio de la red al PC, para que los muestre la aplicación Listen.

La aplicación Listen es una aplicación java que permite escuchar una interfaz de comunicaciones determinada, en nuestro caso el puerto serie COM1, y muestra por pantalla el contenido de los mensajes radio recibidos en formato hexadecimal.

Arrancamos la aplicación Listen en el PC, y posteriormente encendemos los dos nodos. Dado que hemos provisto al mensaje radio que envía el nodo origen con un contador numérico en el campo de datos, lo que vemos es lo siguiente:

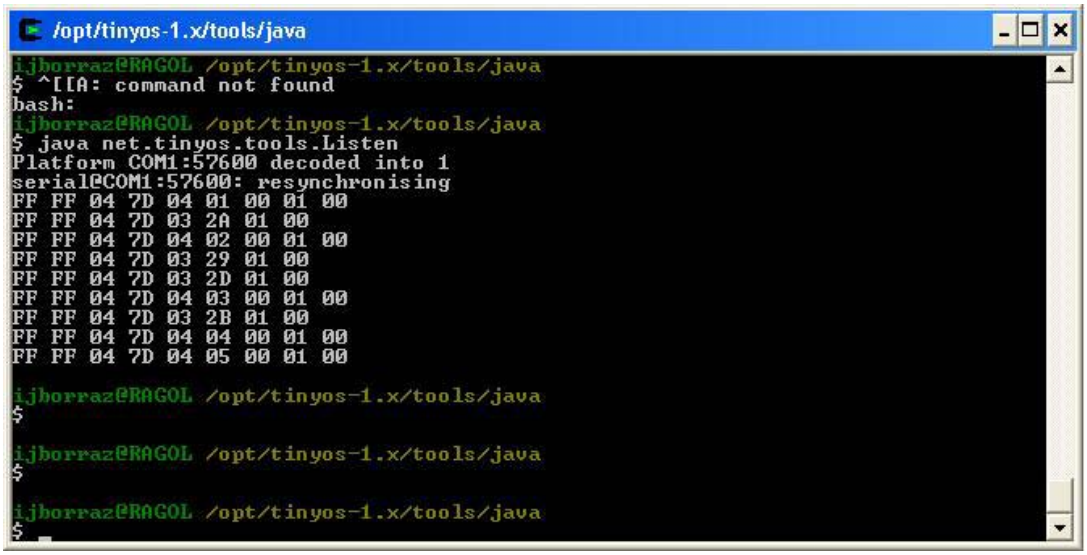


Fig.4.2 Captura de pantalla de la aplicación Listen

El contador numérico se encuentra en el primer byte de datos, éste es el 6º byte empezando por la izquierda.
Los mensajes radio enviados por el nodo1 son los que tienen 9 bytes
Pues bien, si los nodos de forma transparente realizasen reenvío de los mensajes, el nodo2 al recibir el mensaje radio procedente del nodo1 lo reenviaría y en consecuencia a la aplicación Listen del PC, le llegarían los mensajes duplicados.

Como esto no ocurre, nos sirve de demostración práctica para confirmar que los nodos no realizan reenvío de mensajes radio a menos que hayan sido programados para ello.

4.2.3. Generación y recepción de tonos

El código necesario para la interacción de nuestros programas con los diferentes sensores de la placa sensora, se encuentra en el directorio

/opt/tinyos-1.x/tos/sensorboard/micasb

Nótese que el último directorio hace referencia al modelo de placa sensora del que dispongamos, que en nuestro caso es *micasb*.

Una vez allí encontramos los códigos para interactuar tanto con el micrófono, el altavoz, el sensor de luz, etc.

4.2.3.1. Generación de tonos

En este caso los archivos que nos interesan son *Sounder.nc* y *SounderM.nc*.

El archivo *Sounder.nc* es el archivo de configuración del componente y el archivo SounderM.nc es el archivo módulo del componente.

La generación de tonos es una tarea muy sencilla, este componente nos proporciona únicamente dos comandos: uno para empezar la emisión del tono y otro para detener la emisión del tono.

Otra consideración importante es que el tono generado por el altavoz de la placa sensora es de 4,3 Khz. Por lo tanto, los micrófonos receptores deberán ser capaces de aplicar un filtro para detectar el tono a esa frecuencia y avisar a la aplicación del nodo de este hecho.

4.2.3.2. Recepción de tonos por parte del micrófono

El micrófono de la placa sensora tiene dos comandos asociados para su control y un comando asociado para la lectura de la salida binaria procedente del detector de tonos.

El comando que nos interesa más para nuestro propósito es el último, en conocimiento de que el micrófono puede ser configurado como un interruptor de modo que según reciba un tono a una determinada frecuencia generará una salida binaria a 1 (tono no recibido) o a 0 (tono recibido).

Por definición, para el correcto funcionamiento de este comando, el tono recibido debe ser de 4,3 khz.

Con el uso de la interfaz *Mic.nc* basta con que usemos el método readToneDetector() para leer el valor de la salida binaria procedente del detector de tonos (para ello, habremos tenido previamente que configurar el micrófono para trabajar en el modo de detección de tonos).

Sin embargo, otra interfaz, *MicInterrupt.nc* nos hace todavía las cosas más fáciles. Esta interfaz nos proporciona un evento que genera una interrupción cada vez que el micrófono detecta un tono a 4,3 Khz.
Esto nos proporciona dos ventajas muy importantes.
En primer lugar, nos ahorramos la carga de trabajo de la CPU que supondría tener que ir preguntando periódicamente el valor de la salida binaria mediante readToneDetector(), para enterarnos de si el tono se ha recibido o no.

Y en segunda lugar, y tal vez más importante si cabe, dada la aplicación que buscamos implementar, nos permite enterarnos de la recepción del tono tan pronto como éste es procesado (con el comando readToneDetector() nuestro margen de error es tan grande como el período de tiempo que dejemos pasar entre dos ejecuciones consecutivas del comando).

Tiempo de detección

Cuando tenemos el micrófono trabajando en el modo de detección de tonos las etapas electrónicas por las que pasa la señal antes de que se detecte si el tono es válido son las siguientes:

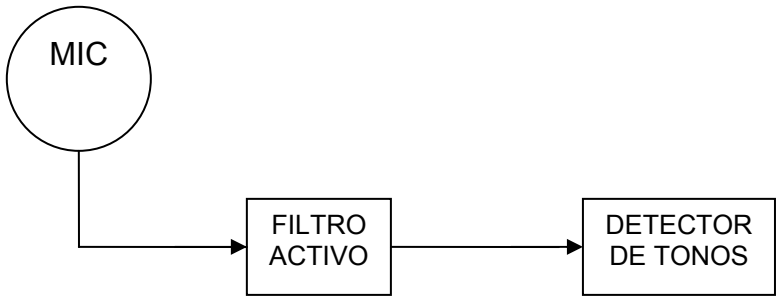


Fig 4.3. Esquema de las etapas de detección de tono

El filtro activo (U2) es un Biquad Active Filter, cuyo elemento central es el integrado MAX4164. En las especificaciones que aparecen en el datasheet de dicho componente, aparecen dos términos que hacen referencia a parámetros temporales:

En operación a 3V:

- Setting time to 0'1% → 50 µs
- Turn-On Time → 20 µs

En operación a 5V:

- Setting time to 0'1% → 70 µs
- Turn-On Time → 40 µs

El elemento central del detector de tonos es el integrado LMC567 CMOS.
En su datasheet nos dicen que el rango de frecuencias de entrada que admite es desde 1 Khz hasta 500 Khz.
También nos avisan de que para que el tono de entrada pueda ser correctamente decodificado, el oscilador debe oscilar a una frecuencia igual al doble de la frecuencia del tono de entrada, y que su valor puede encontrarse mediante el cálculo de la siguiente expresión:

$$F_{osc} = (1 / 1{,}4*Rt*Ct)\ Hz \qquad \textbf{(4.1)}$$

Por tanto, para ir bien, en nuestro caso, la frecuencia del oscilador debería ser de aproximadamente 8 Khz, dado que el tono que genera el altavoz de nuestra placa sensora es de 4,3 Khz (mediante un oscilador piezoeléctrico).

Los valores que aparecen en el datasheet de la placa sensora son: Rt = 25,5 K y Ct = 3,3 nF

Realizamos la comprobación, igualando las siguientes expresiones:

$$(1 / 1{,}4*Rt*Ct)\ Hz = 2 * F_{input}$$

Y el resultado es suficientemente satisfactorio:

$$8{,}48 \approx 8{,}6$$

Comprobamos pues que en la construcción de la placa sensora ya han sido utilizados los valores óptimos de Rt y Ct para la correcta decodificación del tono generado por el altavoz.
En ninguno de los dos datasheet (ni en el del componente LMC567 ni en el de la placa sensora) nos dan ningún valor orientativo de tiempos de retardo provocados por esta etapa.

4.3. Implementación del código nesC

La implementación del código nesC es posiblemente la parte del proyecto a la que se tuvo que dedicar más tiempo.

En el presente apartado lo que se pretende es describir las directrices básicas en que se basó la tarea de programación y dar consciencia de cómo se fueron añadiendo nuevas funcionalidades poco a poco para llegar a cumplir los requisitos del mecanismo de localización.
También se han remarcado algunas optimizaciones del código, aquellas más relevantes por su directa repercusión en el funcionamiento del mecanismo.

4.3.1. Aplicación de generación de tonos

En primer lugar se programó una aplicación muy sencilla en que cada X tiempo periódico, marcado por el vencimiento de un temporizador, el altavoz emitía un tono de 4.3 Khz y cambiaba el estado del led rojo del mica2.
El cambio de estado del led rojo fue programado a modo de *breakpoint* por si el temporizador programado era muy pequeño (dificultad de reconocer auditivamente la diferencia entre el fin de un tono y el inicio del siguiente) o por si el volumen del tono era imperceptible.
Ambos casos se dieron en la primera prueba del código.

El vencimiento del temporizador fue reprogramado para que el tono fuese generado y emitido cada 5 segundos, pero no se encontró la forma de modificar el volumen del tono. Este hecho conllevaba problemas serios porque el tono era apenas audible.

Posteriormente se descubrió que este problema era debido a que la ejecución de las ordenes de arranque (Start) y detención (stop) del dispositivo Sounder, estaban únicamente separada por la instrucción “call Leds.redToggle”, y en consecuencia a la velocidad de procesado del mica2, apenas daba tiempo a arrancar el altavoz que ya se le estaba ordenando que se apagase y el tono apenas podía empezar a emitirse.
Mediante una variable boleana se implementó que a cada vencimiento del temporizador el comando ejecutado fuese alternativamente Start y Stop, de este modo el tono se emitía durante el tiempo X que durase el temporizador y luego se detenía durante el mismo tiempo X, y así sucesivamente.

De este modo se solventó la problemática de la duración del tono (es totalmente modificable cambiando el temporizador) y la problemática del volumen (el volumen real es suficientemente alto como para no ser problema para la implementación de nuestro mecanismo de localización).

A continuación, queríamos dotar al conjunto emisor-receptor de la funcionalidad de contar el tiempo (en ticks de reloj del sistema) desde que el tono era generado por la aplicación generadora de tono hasta que era recibido por el mote programado con la aplicación receptora.
Para implementar esta funcionalidad en nuestro sistema emisor-receptor, se pensó en utilizar un mensaje radio como *beacon*, de modo que la aplicación generadora de tono enviase un mensaje radio justo antes de iniciar el tono de 4,3 Khz, y la aplicación receptora iniciase un contador al recibir el mensaje radio y lo detuviera cuando recibiese el tono de 4,3 Khz.
La idea inicial de nuestra modificación era que el vencimiento del temporizador llamase al fragmento de código que ejecuta la confección y envío de un mensaje radio y fuera éste al finalizar el que llamase al código de inicio o parada de generación de tono ya existente.
Para ello, y habiendo realizado los diferentes capítulos del tutorial sobre TinyOS, la primera idea que surgió fue utilizar la misma interfaz que utilizaba la aplicación *CntToRfm*, que enviaba en mensajes radio los diferentes valores que adquiría un temporizador.
La aplicación *CntToRfm* utiliza la interfaz *IntOutput* que proporciona el componente *IntToRfm* para realizar la tarea de enviar el mensaje radio.
Observando esta interfaz vemos que nos proporciona dos funciones, un comando para programar el envío en sí (que es la que nos proporciona *IntToRfm*) y un evento a programar por el código que utilice dicha interfaz.
Esta última función, que se ejecuta automáticamente cuando se genera el evento de finalización de envío es acorde a nuestras expectativas, pues precisamente se buscaba que en cuanto se hubiese enviado el mensaje radio, se ejecutase el código ya programado de generación de tono.

Por tanto, se reprogramó el evento de vencimiento de temporizador para que llamase a la función IntOutput.output() que envía el mensaje radio y se reprogramó la función evento IntOutput.outputComplete con el código que ya disponíamos.

Evidentemente en el archivo de configuración se reflejaron los cambios que suponía el uso de la nueva interfaz y se le indicó con que archivo tenía que linkar para que nos fuese proporcionado el código.

En este punto se detuvo el desarrollo del código de esta aplicación porque ya cumplía los requisitos básicos que de ella se esperaban:

- Envío de un mensaje RF cada X tiempo definido por un reloj cíclico

- Envío de un tono de 4,3 Khz exactamente después del envío del mensaje RF anterior.

Sin embargo, si nos fijamos en el código que se desarrolló podremos observar que no tan solo envía un mensaje RF antes de iniciar el tono, sino también justo antes de apagar el tono. Esto es debido a que la alternancia inicio-parada de generación de tono depende directamente del evento de conclusión de mensaje radio, que se envía indistintamente del caso cada vez que vence el temporizador.
El efecto negativo de esta situación sobre el funcionamiento de la aplicación no era crítico, pero sí indeseable por dos motivos principalmente:

- Estábamos generando el doble de mensajes radio necesarios, por tanto ocupando el medio radio el doble de tiempo necesario y provocando un gasto energético en el mote que contenía esta aplicación doble e innecesario (si recordamos, la optimización de los recursos energéticos sí es un tema crítico en este tipo de redes).

- Mucho menos crítico pero también importante era el hecho de que el PC receptor de mensajes y encargado de su proceso debía procesar el doble de los mensajes necesarios.

Para paliar esta situación se modificó el código de la forma siguiente:

Fragmento de GeneraTonoM.nc

```
event result_t Timer.fired() {

     estado = !estado;
     if(estado) {
     count++;
     return call IntOutput.output(count);
     } else{
      call Leds.redToggle();
      call SounderControl.stop();
    }
 }
```

```
event result_t IntOutput.outputComplete(result_t success)
{
  if(success == 0) count --;

    call Leds.redToggle();
    call SounderControl.start();
```

Fragmento de GeneraTonoM.nc

Con estas modificaciones solventamos el problema anterior del siguiente modo:

- Ahora la variable utilizada para generar alternancia se encuentra en la función que trata el evento de vencimiento de temporizador, generando ya no alternancia entre inicio-parada de generación de tono, sino alternancia de envío de mensaje radio.
- Cuando se entre en *if(estado)*, el código realizará la tarea de enviar el mensaje radio y cuando éste concluya su envío con éxito (atención: IntOutput.outputComplete nos asegura el envío correcto, NO la recepción correcta) se procederá al inicio del tono.
- La siguiente vez que venza el temporizador, el código entrará en el *else*, de modo que sin previo envío de mensaje radio procederá al apagado del altavoz que esta generando el tono.
- El código realizará esta alternancia hasta que la aplicación finalice (dadas las características del sistema esto sólo ocurrirá cuando se apague el mote que contenga el código de la aplicación).

Por último, se comprobó que con un solo temporizador no era posible conseguir la situación óptima de X tiempo de descanso y Y tiempo de duración del tono en que X fuese mucho más grande que Y.
Con la estructura anterior, si poníamos un temporizador de 10 segundos, era ésta la duración tanto del período de descanso como del tiempo de duración del tono.
Por un lado, deseábamos un período de descanso significativo para no saturar la red de mensajes radio, por las repercusiones que esto tiene en una red inalámbrica donde el medio es compartido y por las motivaciones inherentes a redes de sensores, cuantos menos mensajes menos gasto energético.
Por otro lado, se observó que tonos de larga duración provocaban la falsa percepción, por parte del detector de tonos, de varios tonos emitidos y en consecuencia, el código vinculado al evento de recepción de tono se ejecutaba varias veces para una sola recepción.
Se hizo uso de la característica de nesC que permite instanciar varias veces una misma interfaz para disponer de dos *Timer* diferentes.
En el código lanzado por el envío correcto de mensaje radio (el mensaje radio se envia siguiendo la alternancia de la modificación anterior) se introdujo una llamada al segundo temporizador. En este caso, no era un temporizador periódico, sino simple, el temporizador sólo contaba una vez y posteriormente se ejecutaba el código de su vencimiento.

Así pues, se reestructuró el programa de modo que el código vinculado al envío correcto de mensaje iniciaba el tono e iniciaba el segundo temporizador (puesto a 1 segundo). Con el vencimiento del segundo temporizador se ejecutaba el comando de detención del tono. Tras 9 segundos (el primer temporizador empieza a contar en cuanto vence) se volvía a iniciar el tono, y así sucesivamente.

4.3.2. Aplicación de recepción de tonos

Seguramente denominar a esta aplicación como sugiere el título es menospreciar su cometido. En un inicio, cuando no se era consciente de todas las funcionalidades que debería proporcionar tal vez fuese un nombre justo, pero la aplicación final realiza muchas más funciones.

La aplicación final se encarga de recibir el mensaje radio de sincronización que proviene de la aplicación generadora de tono y posteriormente de detectar el tono.
Cuando ocurre cada uno de estos eventos captura el valor del reloj hardware Systime, y posteriormente realiza una resta para obtener el valor en ticks de reloj que supone la diferencia temporal entre un evento y el otro.
Una vez ha conseguido este valor lo envía mediante un mensaje radio a la red para que sea capturado por el mote *Base* que conectado a la placa de desarrollo permite que los mensajes intercambiados en la red inalámbrica sean recibidos y visualizados en el ordenador mediante la aplicación Listen.

La programación de esta aplicación supuso el estudio con detenimiento del código de las interfaces, *Mic* y *MicInterrupt*.
En el primer código escrito, se intentaba utilizar únicamente la interfaz *MicInterrupt* para gestionar todas las funcionalidades que se necesitaban del micrófono.
Para ello se tomó la implementación de esta interfaz que proporciona el componente *MicC.nc* (que también proporciona los métodos de la interfaz *Mic*) y se realizó paso por paso la configuración del micrófono, desde la especificación de en que modo se quería trabajar (método MuxSel) y la ganancia del micrófono.
Como ya se explicó en informes anteriores, la gran ventaja de las funciones que proporciona *MicInterrupt* consiste en que no debe preguntarse el valor de la salida del detector, sino que es éste quien cuando recibe un tono de 4 Khz envía una interrupción al programa principal, lo cual significa una mejora en cuanto al tiempo de respuesta.
Después de realizar la implementación la aplicación no funcionaba y finalmente se encontró que el código fuente del componente *MicC* tenía un error. El error consistía en que en el comando *init* se deshabilitaba el uso de la interfaz *MicInterrupt*.

Para solucionar esto, se creó un componente *MicCBis* de modo que sí se habilitase el interruptor en la inicialización.

MicMBis.nc

```
command result_t StdControl.init() {

       call ADCControl.bindPort(TOS_ADC_MIC_PORT, TOSH_ACTUAL_MIC_PORT);
       TOSH_MAKE_MIC_CTL_OUTPUT();
       ..
       call MicInterrupt.enable();
       return rcombine(call ADCControl.init(), call PotControl.init());
}
```

Los siguientes pasos a realizar eran dos:

4.3.2.1. Funcionalidad contador

El principio de esta funcionalidad ya se ha explicado en el apartado anterior cuando se explican las modificaciones que tuvieron que realizarse en el código de la aplicación generadora de tono.
Las modificaciones a realizar en este código ya no eran tan evidentes como en la aplicación generadora, ya que la estructura del código no es tan lineal como en el anterior debido a que depende mucho del orden en como le llegue los eventos exteriores.

La ideal inicial era implementar un código que capturase y procesase el mensaje radio de modo que la recepción de éste llamase al comando de captura del valor del temporizador del sistema, para posteriormente, en la función *MicInterrupt.toneDetected()* que se ejecuta tan pronto como se detecta el tono, capturar nuevamente el valor del temporizador. La resta entre estos dos valores nos daría el tiempo de vuelvo del tono en ticks de reloj.
En una primera aproximación se utilizó un código ya desarrollado para la captura y procesado del mensaje radio, para ello se investigó que interfaz utilizaba para ello la aplicación *RfmToLeds*, código de ejemplo que se encargaba de coger los mensajes radio generados por *CntToRfm* con el valor del contador y mostrar este valor en los leds.
Las funcionalidades que necesitábamos estaban divididas entre dos códigos, ya que el análogo a *IntToRfm*, *RfmToInt*, no proporciona la interfaz *IntOutput* (la que procesa el mensaje) pero sí nos linka con la implementación de comunicación *GenericComm* que se encarga de capturar el mensaje radio. En este caso, la interfaz *IntOutput* nos la proporciona el código *IntToLeds*.

Dado que este código proporcionaba una función *IntOutput.output()* diseñada para su propósito (mostrar los valores de un contador enviados por radio en los leds), no nos servía, y se optó por reescribirla manteniendo intacto el resto, en un nuevo componente *IntToLedsBis*. La función se reprogramó de modo que al recibir un mensaje radio se encendiese el led amarillo.

Del mismo modo, el uso de la función *IntOutput.outputComplete()* que realizaba el archivo *RfmToInt* (únicamente devolver SUCCESS) no cumplía con la funcionalidad que nosotros queríamos darle, por tanto lo que hicimos fue utilizar la interfaz *IntOutput* en nuestro código para sobrescribir la mencionada función evento.
Para la captura del valor del temporizador utilizamos la función de la interfaz Systime encargada de ello.

4.3.2.2. Envío al ordenador

Nuestra pretensión era conseguir enviar el valor de este contador mediante un mensaje radio al PC, ya que es este dispositivo el que queremos que, una vez recibidos los valores tomados por los diferentes realice el cálculo que nos permita conocer la posición relativa del mote que envía el tono respecto de los motes que tiene alrededor.

En la implementación de esta nueva funcionalidad básica es muy importante el uso adecuado de la palabra reservada *atomic* en todas aquellas operaciones en que se utilice la variable *ticks* que lleva el valor del contador.
La importancia de proteger este valor se debe a que los tonos de 4,3 Khz se reciben de forma periódica y tenemos que asegurar que mientras se está formando el mensaje radio para enviar una medición al PC, una nueva medición no nos sobrescriba el valor tomado en *ticks*, ya que esto supondría la pérdida de una medida por un lado y el consiguiente cálculo erróneo en el PC de la posición del mote.
Centrémonos ahora en el modo como se consiguió implementar la funcionalidad.

En un primer intento, se recurrió al uso de la implementación de la interfaz *IntOutput* que hacia el componente *IntToRfm*, ya que esta interfaz nos había permitido implementar de forma sencilla y mediante reutilización de código ya existente, la funcionalidad de transmitir un mensaje radio en nuestra aplicación generadora de tono.
El primer problema encontrado fue que nuestra aplicación receptora ya hacía uso de la interfaz *IntOutput* para proporcionar la funcionalidad de procesado del mensaje radio que se recibía proveniente de la aplicación generadora.
Sin embargo, a la práctica esto no supone problema ya que mediante la capacidad de renombrar interfaces que nos proporciona el lenguaje nesC (1), podemos utilizar de nuevo la interfaz *IntOutput* simplemente haciendo constar su nuevo nombre tanto en el archivo de configuración como en el archivo de módulo. En este caso se renombró a *IntBase*.

---

[1] útil en casos como estos, donde se debe utilizar la misma implementación o implementaciones diferentes de una misma interfaz con propósitos distintos y no queremos que se confunda código, ya que sin esta capacidad tendríamos tantas funciones evento definidas con el mismo nombre como usos de la interfaz y el compilador no sabría diferenciar cuando se debe usar una u otra.

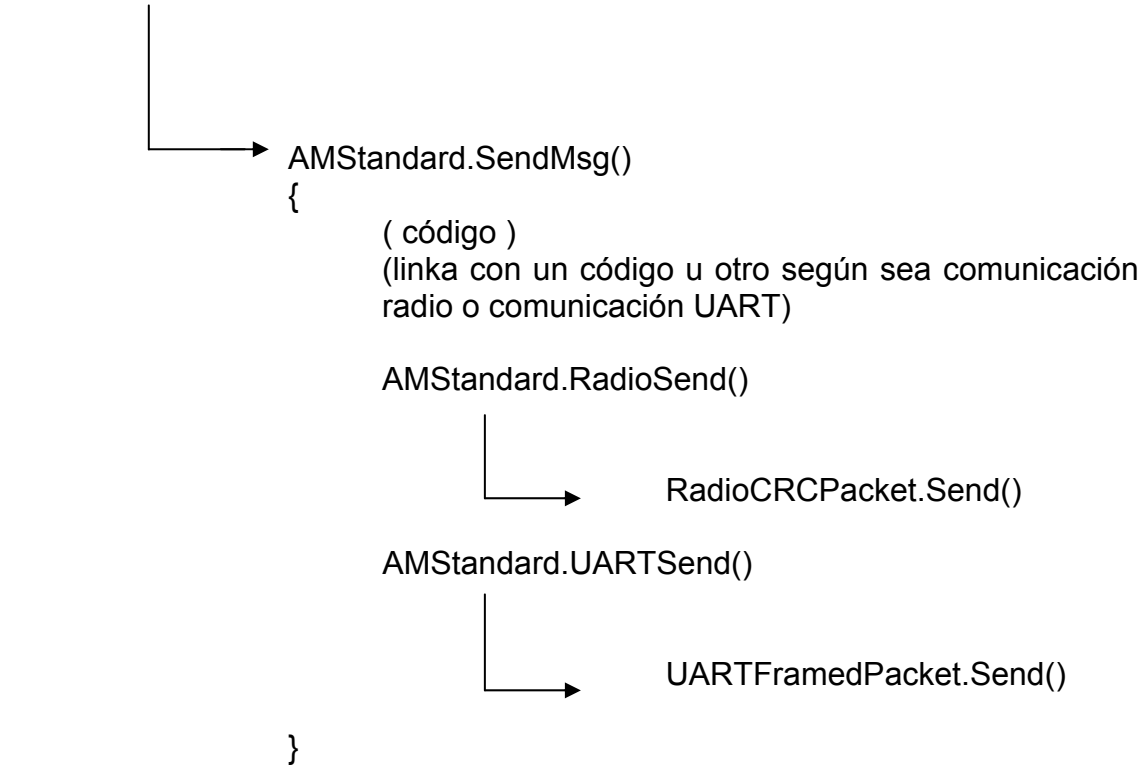
De este modo teníamos nombres de funciones evento distintas en cada caso: en el que ya teníamos seguirá siendo IntOutput.outputComplete() , y en el nueva uso será IntBase.outputComplete().

Optimización de contador y envío a pc

Se optó por estudiar la estructura de los códigos e interfaces de más bajo nivel relacionados con la transmisión y recepción de mensajes radio.

La interfaz genérica de comunicación que utilizan todos los códigos de ejemplo es *GenericComm*, las funcionalidades de envío y recepción (SendMsg y ReceiveMsg definidas en interfaces del mismo nombre) las linka del archivo *AMStandard*, que implementa parte del código de envío y recepción para luego linkar parte del código de envío y recepción de otros archivos según se tenga que implementar la comunicación vía radio o vía UART.

La estructura es la siguiente:



La estructura es la misma para la interfaz *ReceiveMsg*.

Conociendo la estructura se programó una solución diferente basada directamente en el uso de la implementación de la interfaz *SendMsg* que proporcionaba el archivo *GenericComm* (y que ya hemos visto de dónde toma él los códigos para implementar la comunicación en la estructura anterior).
Esta opción nos daba, además de mayor control y conocimiento sobre las funciones que se ejecutaban, la posibilidad de cambiar ciertos parámetros que utilizando la interfaz *IntOutput* nos venían impuestos por su implementación.

De este modo se pudo adaptar la estructura comprendida en el campo "data" del mensaje radio (TOS_Msg definido en AM.h) que utilizaba *IntOutput*, que era la estructura *IntMsg* definida en *IntMsg.h*, por una nueva que llamamos *IntMsgBB.h.*

La decisión de adaptarla se tomó tras comprobar que la estructura *IntMsg* no nos servía.
Esta estructura de mensaje utilizaba 2 bytes (uint16_t) para almacenar los datos a transmitir (a parte de 2 bytes utilizados para transmitir la ID del nodo), sin embargo en nuestro caso la variable *ticks* estaba definida como uint32_t (debido al uso del comando Systime.getTime32() para obtener el valor del reloj) y por lo tanto necesitábamos definir un campo de datos de 4 bytes capaz de contener esta variable.

4.3. Implementación de la aplicación Java

El propósito de esta aplicación es capturar los mensajes provenientes de los nodos receptores de la red para procesar el valor de ticks de reloj que contienen.
Este proceso consiste en la conversión a distancia en metros para posteriormente poder aplicar un algoritmo de triangulación que determine la posición del nodo ilocalizado (el que genera el tono).
Además se le ha dotado de una interfaz gráfica amigable para la visualización de los datos de salida.

Nuestra aplicación se ha basado en la aplicación existente Listen que ya proporciona la captura de mensajes radio.

4.3.1. Aplicación MyListen

Esta aplicación se ha escrito tomando como base la clase Listen y el código es el mismo hasta la captura de los mensajes, en este punto es donde la nueva clase MyListen añade todas sus funcionalidades extra.
Esta clase va a ser el centro de nuestra aplicación ya que va a ser el método main, y a su alrededor van a crearse otras clases de las que MyListen va a servirse para implementar todas sus funcionalidades.

Vamos a crear un modelo nuevo de tabla mediante Swing (2) para implementar una interfaz gráfica que muestre los resultados deseados al usuario final. Este modelo se encuentra en la clase ModeloTabla.
La interfaz gráfica va a ser una clase de la que crearemos un nuevo objeto en nuestro main principal nada más empezar, la clase que nos va a implementar la interfaz gráfica ha sido denominada MyTable.
A medida que lleguen los datos de la red y se pueda realizar la triangulación las filas de la tabla se irán actualizando para mostrar la información recogida y procesada.

Se ha creado un modelo de tabla de 20 filas, cuando ya se han rellenado las veinte filas la siguiente visualización de datos sobrescribe los datos de la primera fila.
La tabla muestra los valores de *ticks* recogidos por 3 nodos y la posición estimada del nodo ilocalizado que se obtiene después de realizar la triangulación
Para evitar los problemas de interpretación de resultados derivados de esto se ha añadido en la tabla una quinta columna donde se muestra la hora en que se procesa la información mostrada.
De este modo aunque se sobrescriban filas la hora permite discernir al usuario en que orden van las medidas que actualmente visualiza la interfaz gráfica.

Una vez explicado el modo de visualización de los datos vamos a centrarnos en el proceso interno de nuestra aplicación.
Los mensajes radio de TinyOS, programados en nesC con el nombre de estructura TOS_Msg, tienen la siguiente estructura:

- 5 bytes de cabecera:
  - Dirección: 2 bytes
  - Tipo AM: 1 byte
  - Grupo: 1 byte
  - Tamaño del campo de datos: 1 byte

- N bytes del campo de datos

Como hemos visto en secciones anteriores, el mensaje radio que transmiten los nodos receptores conteniendo el valor de *ticks* tiene una longitud del campo de datos de 6 bytes: 4 bytes para almacenar la variable *ticks* y 2 bytes donde se especifica la ID del nodo.
Por tanto, la longitud de estos mensajes radio va a ser de 11 bytes. Por el contrario, la longitud del mensaje radio de sincronismo sigue la estructura predefinida *IntMsg* que consta de 9 bytes.
De este modo, vamos a utilizar la longitud del mensaje para discernir que mensajes nos interesa procesar, que son en este caso los de los nodos receptores.

La aplicación Listen nos encapsula cada mensaje radio en un array de bytes.
Si el paquete pertenece a uno de los motes receptores llamamos a un método encargado de capturar el valor de los ticks de reloj del temporizador.
Este método va a tener que concatenar los bytes que contienen la información de ticks en orden inverso debido a que el envío de los campos de datos por el medio de transmisión es *little endian*.
Para aumentar la modularidad y clarificar el código, se ha decidido crear una clase aparte llamada *CapturaDatos* que implementará este método. A su finalización, almacenamos en una variable el número de ticks ya transformado a decimal que contenía el paquete recibido.

---------- (2) paquete de java que integra todos los componentes necesarios para crear interfaces gráficas. Es el sucesor del antiguo paquete java.awt, incorpora componentes de awt y los hace mejores y más potentes, además de añadir muchos componentes nuevos. ----------

Vamos a considerar que en nuestro escenario tenemos un mote emisor de tono (de quien se desconoce la posición) y tres motes detectores de tono (que toman medida de distancia y la envían al pc).

Si recordamos los dos últimos bytes de datos, bytes 10 y 11 en este tipo de mensajes, contienen la ID del dispositivo. Esta ID puede ser asignada en el momento de subir el código al mote y en este caso se supone que se han asignado las ID 1, 2 y 3 a los tres motes detectores de tono.
Mediante una sucesión de "*if*" se evalúa a que mote pertenece la medida recibida y si todavía no ha llegado ninguna de él (contemplamos la posibilidad de recibir 2 o más medidas de un mismo mote antes de recibir la primera medida de otro de ellos y en este caso debemos descartar esa segunda medida) se asigna la medida a otra variable.
Cuando ya hemos recibido tres medidas válidas, una perteneciente a cada uno de nuestros motes detectores de tono, ya podemos llamar al método "Triangulation" para realizar la triangulación y encontrar la posición del mote que buscamos.
Nuevamente hemos creado una clase nueva, denominada MathLocalization, que contiene todos los cálculos matemáticos relacionados con la triangulación, de este modo se contribuye a mejorar la modularidad y a clarificar el código.
El algoritmo de triangulación utilizado se ha realizado a imagen y semejanza del propuesto por [24].

Una vez disponemos de la posición, llamamos al método AsignarValor definido en nuestra implementación de la interfaz gráfica que muestra los resultados en una tabla, para que actualice los campos de la tabla y se visualice la posición calculada así como los ticks que han dado cada uno de los nodos de referencia.

Un último detalle a mencionar es que todas las clases que toman parte en la actual aplicación: MyListen, CapturaDatos, MathLocalization, MyTable y ModeloTabla se ha definido que pertenezcan a un paquete denominado MyListener.

Se presentan varias capturas de esta aplicación en el capítulo de resultados.

CAPÍTULO 5: RESULTADOS

En este capítulo quinto se van a presentar los resultados obtenidos con las versiones definitivas de las dos aplicaciones: la aplicación generador-receptor programada para los nodos sensores en nesC y la aplicación java de visualización de datos.
Aunque ambas son parte de un mismo mecanismo se ha preferido separar los resultados y las pruebas porque se han evaluado problemáticas distintas.

5.1. Resultados de la aplicación nesC

El escenario de pruebas estuvo compuesto por 3 motes mica2, una placa de desarrollo MIB510 y un ordenador con las características explicadas en el capítulo de entorno de trabajo.
La intención era obtener resultados de las medidas de *ticks* de reloj a diferentes distancias para comprobar cuanto se acercaban o se alejaban de la distancia real entre los motes.
Para ello en el ordenador se ejecutó la aplicación Listen, encargada de mostrar por consola los mensajes radio que le llegaban por el puerto serie COM1 provenientes de la placa de desarrollo.
En la placa de desarrollo un mote conectado a ella ejecutando el código TOSBase era el encargado de capturar los mensajes radio de la red y enviarlos al ordenador mediante cable serie.
Por último, un mote ejecutando la aplicación generadora de tono y otro mote ejecutando la aplicación receptora de tono se situaron encima de la mesa del laboratorio y se fueron alejando uno del otro hasta que se encontró la distancia umbral en que el tono se hacía inaudible para el mote receptor (ya no detectaba el tono).

Para cada distancia medida se han tomado las 10 primeras mediciones realizadas por el mote que calcula en ticks de reloj la diferencia temporal entre la recepción de un mensaje radio y del tono de 4,3 Khz.
Debido a un comportamiento de cuelgue indeseado que ha experimentado la aplicación receptora de tonos y a algunos errores, se han seguido las siguientes pautas para discernir entre mediciones válidas y mediciones desestimadas:

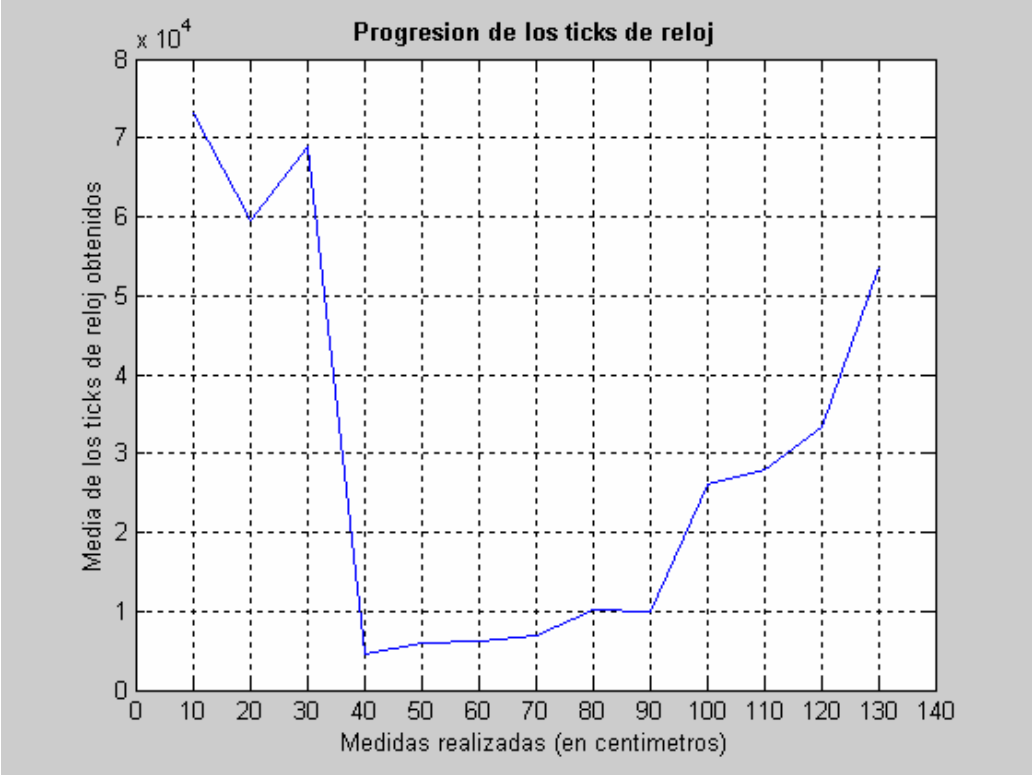
- El orden correcto de aparición de mensajes radio en la aplicación Listen es de 1 mensaje radio proveniente del nodo generador de tono y 1 mensaje radio proveniente del nodo calculador de distancia. Cuando por error de la aplicación programada en este segundo nodo, se recibe más de 1 resultado por tono emitido, los resultados adicionales se desestiman por considerarse resultado de un error.

- Así mismo, se ha procurado en las mediciones realizadas que lo han permitido, no considerar el último mensaje radio de distancia medida, antes del cuelgue de la aplicación, se ha considerado que este último mensaje puede ser erróneo.

Con estos valores transformados a valores en distancia se han realizado gráficas para cada distancia real que permitiesen evaluar el grado de corrección en el funcionamiento del mecanismo. Estas gráficas se han adjuntado en un anexo del presente proyecto.
Para la evaluación de los resultados se utilizarán estas gráficas:

- Una gráfica donde se representa el incremento de ticks de reloj en función de la distancia (se ha realizado la media aritmética de las 10 medidas de cada distancia para tal propósito)

- Y una gráfica comparativa de aquellas distancias cuyos valores medidos eran suficientemente cercanos como para no permitir ver a simple vista con sus gráficas particulares la evolución de la medición realizada por la aplicación.



En la gráfica de progresión de los ticks de reloj se puede observar un comportamiento totalmente anormal hasta la distancia de 40 cm.
En las gráficas particulares podrá visualizarse la gran dispersión y valores muy altos que presentan los valores medidos para las distancias inferiores a 40 centímetros y en base a los que suponemos que por problemas de saturación del micrófono, interferencias entre motes, o problemas de otra índole, el funcionamiento es erróneo para este rango de mediciones.

También se ha pensado como causante de esta situación un problema de velocidad de procesador.

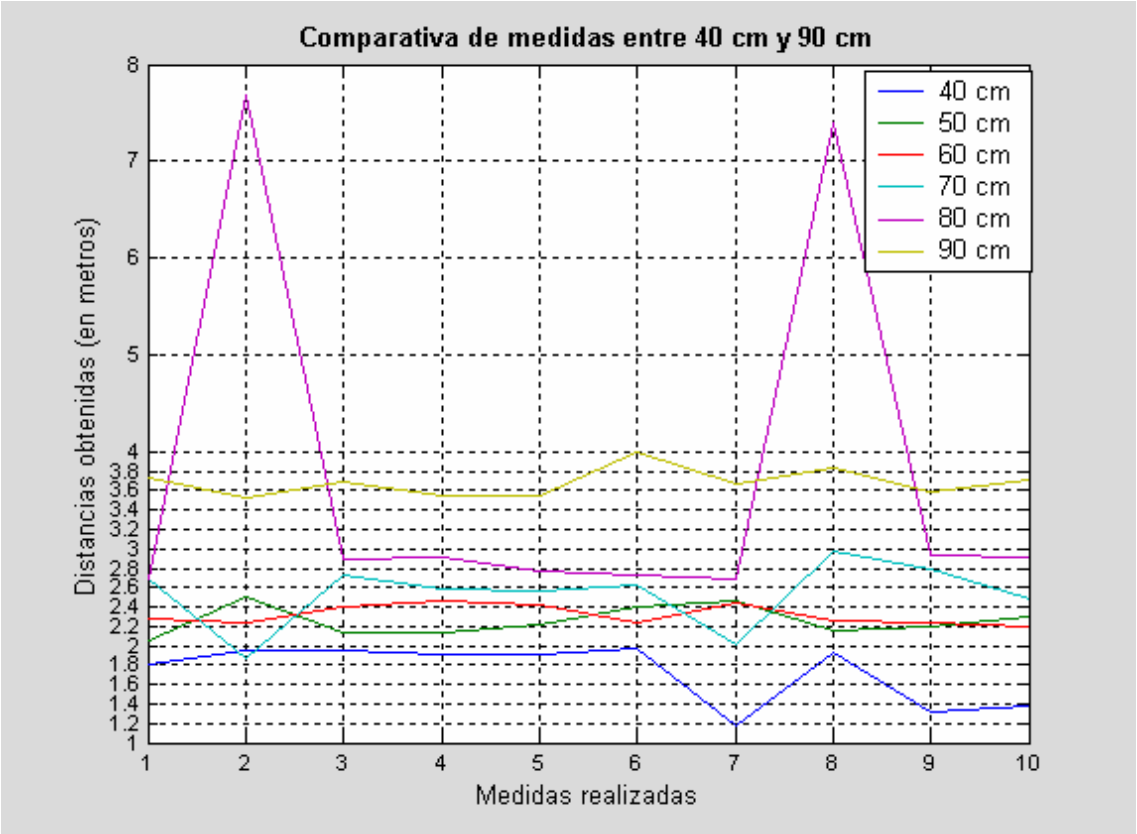
Según se ha visto tras estudiar el modelo de concurrencia de nesC, hay interrupciones (generadas por recepción de eventos) que pueden apartar de la ejecución a código llamado por interrupciones anteriores.
Si recordamos, cuando se notifica la llegada de un mensaje radio en la aplicación receptora, ésta lanza un código donde captura el valor de ticks actual en una variable, *TicksMsg*. Y también tiene un código que se lanza cuando detecta un tono donde se captura el valor de ticks en ese momento en la variable *TicksTone*, y se realiza la resta de estos dos valores (*TicksTone* – *TicksMsg*) para encontrar la diferencia de ticks entre ambos eventos que posteriormente se enviará en un mensaje radio.
Cuando los nodos se encuentran muy cerca es posible que el tono sea detectado antes de que la aplicación termine de ejecutar el código que generó la recepción del mensaje.
¿Que ocurre entonces si la interrupción detectora de tono interrumpe el proceso del código ejecutado por la interrupción detectora de mensaje?
Si ocurre, es posible que ocurra antes de que el código lanzado por la recepción del mensaje tenga tiempo de guardar el valor de ticks en *TicksMsg*.
De este modo, si acabamos de encender los motes, cuando el código lanzado por la recepción del tono ejecute la resta *TicksTone* – *TicksMsg*, *TicksMsg* valdrá 0 y la resta será el valor actual del temporizador (que recordemos empieza a contar cuando se enciende el mote), *TicksTone*.
Si en alguna de las mediciones periódicas por el motivo que sea (el procesador está más libre sería la más factible o retardo de detección de tono) el código lanzado por la recepción del mensaje sí tiene tiempo de terminar, entonces *TicksMsg* tomará un valor correcto y la siguiente ejecución de *TicksTone* – *TicksMsg* nos dará un resultado válido.
Sin embargo, esta situación podría no volver a repetirse hasta X mediciones después. En ese caso, *TicksMsg* seguiría valiendo lo mismo mientras que *TicksTone* se incrementaría cada vez, provocando que el resultado de su resta aumentase progresivamente.

A partir de los 40 centímetros, las mediciones obtenidas no se corresponden con la medida real de distancia pero si siguen un patrón lógico.
Por tanto, su diferencia podría solventarse con una simple calibración.

A medida que distanciamos los motes las mediciones incrementan de forma regular y lineal hasta los 90 centímetros. A partir de 90 centímetros el incremento toma forma aproximadamente exponencial.
Esta forma exponencial puede ser causada porque el detector de tonos tarde más en detectar el tono por motivo de la distancia, además, si tenemos en cuenta que la distancia umbral encontrada en que el micrófono del mote es incapaz de detectar el tono es de alrededor de 1,3 metros, no extraña tanto que a partir de esa medida empiece a costarle la detección.

Cabe destacar para dar más énfasis o credibilidad al párrafo anterior que las medidas se realizaron en el laboratorio en horario de fin de semana, en que únicamente nuestro ordenador estaba encendido y no había ruido ambiental provocado por otras personas u otros dispositivos electrónicos.

Por tanto, nos encontrábamos en las condiciones óptimas para que el micrófono detectase el tono por ausencia de otras señales acústicas que pudiesen interferir el tono.



Con esta gráfica comparativa se intenta dar énfasis al hecho de que las medidas entre 40 y 90 centímetros siguen una progresión lineal en función de la distancia.

Salvo alguna medición excepcional (como los dos picos de las medidas 2 y 8 de la distancia 80 centímetros) se puede observar como el incremento en la medición estimada para cada medida real es de 20 centímetros respecto a 10 centímetros y sobretodo que el mecanismo implementado da idea de distancia (una medición a mayor distancia no da una medición estimada menor que una medición a distancia más pequeña).

5.2. Resultados de la aplicación Java

Ha sido imposible testear el escenario completo porque no se ha conseguido portar las aplicaciones nesC desarrolladas para mica2 a la plataforma mica2dot.

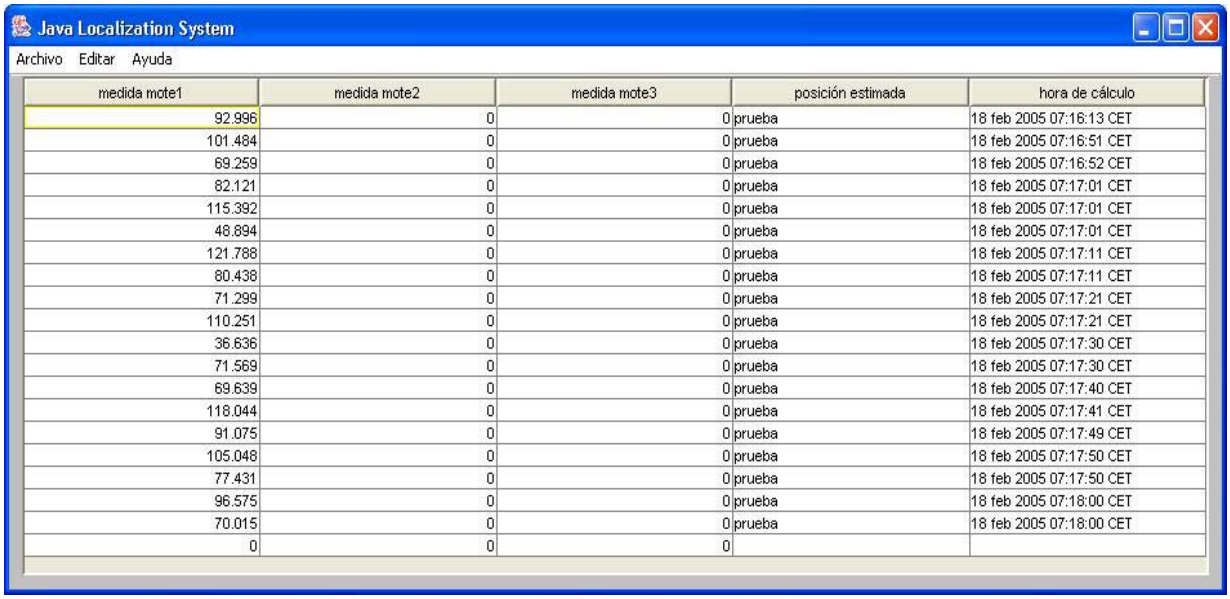
La aplicación java, por requisitos del algoritmo de triangulación, necesita un mínimo de tres medidas que provengan de tres motes diferentes para obtener un resultado satisfactorio.
Se disponía únicamente de dos placas sensoras para plataforma mica2 de modo que únicamente se ha conseguido una medición de ticks (la otra placa la necesitaba el mote ilocalizado para generar el tono).

Así pues, no se han podido evaluar al cien por cien las funcionalidades de la aplicación java.

Para demostrar el correcto funcionamiento de todas sus partes se diseñaron códigos de prueba que funcionasen bajo parámetros introducidos por consola que se han añadido en un anexo del presente proyecto.
Todos esos códigos de prueba están debidamente comentados para su comprensión.

En este capítulo vamos a evaluar los resultados de capturas de la aplicación en su funcionamiento limitado debido al escenario, y en un código de prueba que demuestra todo su potencial ya que únicamente cambia en el hecho de que los 3 arrays de bytes que debería capturar de COM1 le son introducidos por consola.

De este modo, en su funcionamiento limitado únicamente vamos a poder evaluar la correcta presentación de la interfaz gráfica y mediante la visualización de los datos la correcta actualización de la tabla y proceso (nuevamente limitado) de los valores capturados de la red sensora.

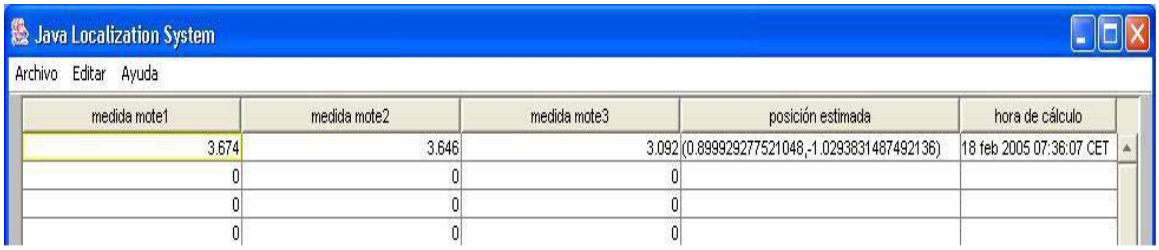


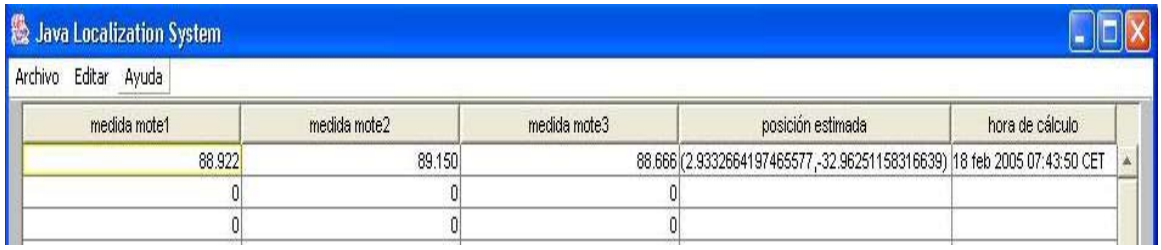
medida mote1	medida mote2	medida mote3	posición estimada	hora de cálculo
92.996	0	0	prueba	18 feb 2005 07:16:13 CET
101.484	0	0	prueba	18 feb 2005 07:16:51 CET
69.259	0	0	prueba	18 feb 2005 07:16:52 CET
82.121	0	0	prueba	18 feb 2005 07:17:01 CET
115.392	0	0	prueba	18 feb 2005 07:17:01 CET
48.894	0	0	prueba	18 feb 2005 07:17:01 CET
121.788	0	0	prueba	18 feb 2005 07:17:11 CET
80.438	0	0	prueba	18 feb 2005 07:17:11 CET
71.299	0	0	prueba	18 feb 2005 07:17:21 CET
110.251	0	0	prueba	18 feb 2005 07:17:21 CET
36.636	0	0	prueba	18 feb 2005 07:17:30 CET
71.569	0	0	prueba	18 feb 2005 07:17:30 CET
69.639	0	0	prueba	18 feb 2005 07:17:40 CET
118.044	0	0	prueba	18 feb 2005 07:17:41 CET
91.075	0	0	prueba	18 feb 2005 07:17:49 CET
105.048	0	0	prueba	18 feb 2005 07:17:50 CET
77.431	0	0	prueba	18 feb 2005 07:17:50 CET
96.575	0	0	prueba	18 feb 2005 07:18:00 CET
70.015	0	0	prueba	18 feb 2005 07:18:00 CET
0	0	0		

Para esta captura hemos configurado el escenario de pruebas de la aplicación nesC pero en el ordenador en lugar de ejecutar la aplicación Listen, hemos ejecutado nuestra aplicación MyListen.

Nuestra aplicación lanza su interfaz gráfica y a medida que le llegan los mensajes radio oportunos (únicamente los de 11 bytes como se explica en el capítulo de desarrollo) los procesa y actualiza la tabla.
Nuestra aplicación no resta ninguna funcionalidad a la aplicación Listen, únicamente las suma, pues el código original sigue funcionando. Además de ver los resultados en la interfaz gráfica, en consola siguen apareciendo las cadenas de mensaje en hexadecimal.

Para poder realizar esta prueba se tuvo que modificar levemente el código de la aplicación MyListen. Debido a la imposibilidad de realizar el cálculo de triangulación porque sólo podía capturarse una medida, se optó por mostrar un String en el campo de la tabla definido para tal efecto. De este modo se comprueba que el campo funciona correctamente, ya que la posición estimada también se le pasaba como String.
Por último podemos comprobar como en la quinta columna aparece la hora en que se capturó la medida. Como se puede comprobar, por ejemplo en las dos últimas filas, tenemos 2 medidas a las 07:18:00, debido a los errores de medidas duplicadas de la aplicación nesC de los que hablamos al inicio de este capítulo.





Estas dos capturas han sido obtenidas con el código de prueba anteriormente mencionado.
Toda la estructura de la aplicación es la misma que la aplicación final, la única diferencia se encuentra en el hecho de que los mensajes radio se simulan introduciéndolos por teclado cuando se ejecuta la aplicación.
Podemos comprobar como se realiza el cálculo de triangulación y como su resultado se muestra con toda precisión en el campo de "posición estimada" de la tabla.
Se han mostrado dos capturas para ejemplificar que funciona para todo rango de valores, en el primer ejemplo son medidas de unos 3.000 ticks y en el segundo de unos 90.000.
No obstante, no toda medida de ticks es posible.

Para poder implementar el algoritmo de triangulación se deben conocer las posiciones físicas de los tres nodos de referencia. Para ello, en la aplicación java se han predefinido como (0,0), (1,1) y (2,0) (intentando que fuesen bastante reales a las posiciones que habríamos podido distanciarlos de haber tenido el escenario completo).
Si se intenta introducir una medida de ticks muy diferente entre los nodos, es decir, por ejemplo el mote 1, valor 3.000 ticks y el mote 2 valor 17.000 ticks, el resultado de la posición estimada será (NaN, NaN) donde NaN significa Not A Number.
Esto es debido a que conociendo sus posiciones, el algoritmo de triangulación no puede computar mediciones absurdas. Es absurdo que un nodo que se encuentra a 1 metro de otro nodo detecte una distancia en ticks respecto a un tercer nodo 4 o 5 veces mayor que el otro nodo de referencia. Los ticks de diferencia deberían ser aproximadamente los que corresponden a un 1 metro, sin embargo el algoritmo de triangulación tampoco es tan severo y permite diferencias mayores.

CAPÍTULO 6: CONCLUSIONES Y VÍAS FUTURAS

El presente proyecto ha servido para evaluar la capacidad de los motes mica2 de Crossbow de implementar un mecanismo de localización basado en tiempo de vuelo de señal de sonido.
Aunque no se ha podido probar el mecanismo en un escenario completo, los resultados obtenidos demuestran que el mecanismo de localización implementado es capaz de dar idea de distancia y en consecuencia idea de localización en los márgenes comprendidos entre 40 centímetros y 130 centímetros.

Además se ha conseguido realizar una aplicación java que se sirve de las librerías de TinyOS y por tanto puede integrarse en el sistema operativo sin ningún problema de compatibilidad con la actual *release*. Esta aplicación permite el testeo del algoritmo de triangulación de un escenario con 3 nodos de referencia, sin embargo, la mayor parte de su código es fácilmente reutilizable, hay métodos y clases especialmente diseñados para habilitar su reutilización, y el código que no lo es requiere de muy pocas modificaciones para adaptarse a un escenario mayor.

En cuanto a la aplicación realizada para los nodos sensores, la imposibilidad de haber tenido un escenario completo se debe a que el código no es directamente portable entre la plataforma mica2 y la plataforma mica2dot. Esto supone un inconveniente de cara a la uniformidad del código de aplicación en toda una red. Sin embargo el hecho de que a alto nivel los componentes de TinyOS sean los mismos para cualquier plataforma permite asegurar que las funciones de red se entiendan entre ellas.

Por otro lado el constante cuelgue de la aplicación receptora ha provocado mucho tiempo de retraso en la conclusión del presente proyecto. Se estuvo prácticamente un mes y medio parado intentando discernir el porqué de tal comportamiento sin llegar a ningún resultado concluyente.

La vía futura más inmediata pasaría por la mejora de la aplicación receptora para que no se diese lugar la situación anormal comprendida entre les 0 centímetros y los 40 centímetros, para ello sería necesario estudiar los sucesos causantes que aquí se suponen.

La aplicación Java también podría ser muy mejorada con la incorporación de una interfaz gráfica adicional que dibujase los nodos de referencia y dibujase la posición estimada del nodo ilocalizado. Sería mucho más vistoso y amigable para el usuario final que una tabla de resultados.

Por otro lado, muchos de los grupos desarrolladores de sistemas de localización basados en los tonos de las plataformas Crossbow, han abandonado esta línea de investigación substituyéndola por el mismo acercamiento pero utilizando el generador de ultrasonidos que contiene la nueva placa sensora.

El código realizado para la aplicación de los motes debería cambiar muy poco para ser adaptado al uso de ultrasonidos, y sin duda éste es ya ha empezado a ser el presente y es sin duda el futuro del desarrollo de mecanismos de localización basados en motes de Crossbow.

Debido a que el mecanismo desarrollado está principalmente basado en medidas de alcance, convendría rediseñar el mecanismo para que se acercara más a las técnicas *range-free*, mucho más indicadas para redes de sensores.
En este sentido, el mecanismo propuesto DV-Hop muestra un concepto muy interesante para enfocar la localización.

Bibliografía

[1] Chee-yee, C. and Kumar, S.P., "Sensor networks: evolution, opportunities, and challenges", *Proceedings of the IEEE*. 91 (8), 1247-1256 (2003).

[2] Akyildiz, I.F. *et al*, "A survey on sensor networks", *IEEE Communications Magazine*. (8), 102-114 (2002).

[3] Hoblos, G. Staroswiecki, M. and Aitouche, A., "Optimal Design of Fault Tolerant Sensor Networks", *IEEE Int'l. Conf. Cont. Apps.*, Anchorage, AK, 467–72 (Sept 2000)

[4] Bulusu *et al.*, "Scalable Coordination for Wireless Sensor Networks: Self-Configuring Localization Systems", *ISCTA 2001*, Ambleside, U.K., (July 2001).

[5] Shih, E. *et al.*, "Physical Layer Driven Protocol and Algorithm Design for Energy-Efficient Wireless Sensor Networks", *Proc. ACM MobiCom '01*, Rome, Italy, July 2001, pp. 272–86

[6] Woo, A. and Culler, D. "A Transmission Control Scheme for Media Access in Sensor Networks", *Proc. ACM MobiCom '01*, Rome, Italy, July 2001, pp.221–35.

[7] Kahn, J.M. Katz, R.H. and Pister, K.S.J., "Next Century Challenges: Mobile Networking for Smart Dust", *Proc. ACM MobiCom '99*, Washington, DC, 1999, pp. 271–78.

[8] Hightower, J. and Borriello, G., "Location systems for ubiquitous computing" *Computer*. 34 (8), 57-66 (2001).

[9] Barton, J. and Kindberg, T., "The cooltown user experience". *Technical Report.* 22 (2001).

[10] Harter, A. *et al*., "The anatomy of a context-aware application". *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking.* 59-68 (1999).

[11] Priyantha, N.B. Chakraborty, A. and Balakrishnan, H., "The cricket location-support system". *Proceedings of MOBICOM 2000*. 32-43 (2000).

[12] "PulsON Technology: Time Modulated Ultra Wideband Overview" Time Domain Corporation (2001).

[13] Hightower, J. Want, R. and Borriello, G., "SpotON: An indoor 3d location sensing technology based on RF signal strength". UW-CSE 00-02-02, University of Washington, Department of Computer Science and Engineering, Seattle, WA, February 2000.

[14] Bahl, P. and Padmanabhan, V., “RADAR: An in-building Rfbased user location and tracking system”, *Proceedings of IEEE INFOCOM* (2), 775-784 (2000).

[15] Hinckley, K. and Sinclair, M., “Touch-sensing input devices”, *Proceedings of the 1999 Conference on Human Factors in Computing Systems* (1999).

[16] Partridge, K. *et al*., “Fast intrabody signaling”. *Demonstration at Wireless and Mobile Computer Systems and Applications (WMCSA)* (2000).

[17] Want, R. *et al*., “The active badge location system”, *ACM Transactions on Information Systems*. 10 (1), 91-102 (1992).

[18] Want, R. *et al*., “The parctab ubiquitous computing experiment”, Cap. 2 en *Mobile Computing* 45-101, Tomasz Imielinski editor, Kluwer Publishing (1997).

[19] Hills, A., “Wireless andrew”, *IEEE Spectrum*. 36(6), 49-53 (1999).

[20] He, T., “Range-Free Localization Schemes for Large Scale Sensor Networks”, *MobiCom ’03*, September 14-19, 2003, San Diego, California, USA.

[21] Bulusu, N. Heidemann, J. and Estrin, D., “GPS-less Low Cost Outdoor Localization for Very Small Devices”, *IEEE Personal Communications Magazine*. 7(5), 28-34 (2000).

[22] Niculescu, D. and Nath, B., “DV Based Positioning in Ad Hoc Networks”, *Telecommunication Systems* 22 (1–4), 267–280 (2003).

[23] Nagpal, R., “Organizing a Global Coordinate System from Local Information on an Amorphous Computer”, *A.I. Memo 1666*, MIT A.I. Laboratory, August 1999.

[24] Iyengar, R. and Sikdar, B., “Scalable and Distributed GPS free Positioning for Sensor Networks”, *IEEE* (2003).

[25] Gay, D. *et al*., “The nesC language: A Holistic Approach to Networked Embedded Systems”. *Intel Research Berkeley,* Intel Corporation (2002).

Enlaces web

Lesson 1: Introduction to TinyOS: Components, Composition, and Basic Use of Commands and Events.
http://www.tinyos.net/tinyos-1.x/doc/tutorial/index.html

Crossbow Technology, Inc.
www.xbow.com

ANEXOS

ANEXO 1: ESTADO DEL ARTE EN REDES DE SENSORES

La pila de protocolos utilizada por el sink y los nodos sensores se muestra en la figura 1.1.

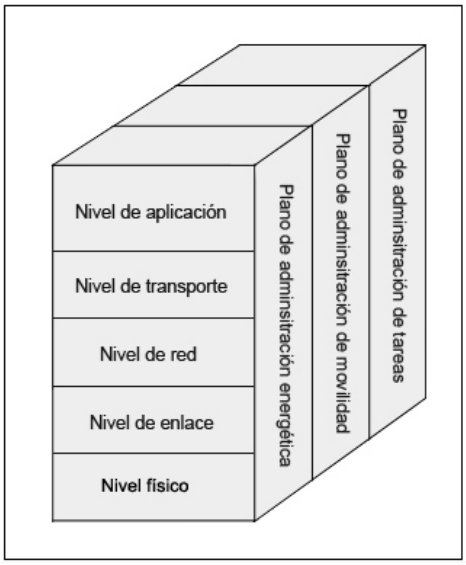


Figura 1.1. Pila de protocolos

Los planos de administración energética, de movilidad y de tareas ayudan a los nodos sensores a coordinar la tarea de percepción y a disminuir el consumo energético global.

El plano de administración energética gestiona como un nodo sensor utiliza su energía. Por ejemplo, el nodo sensor puede apagar su receptor después de recibir un mensaje desde uno de sus nodos vecinos. Esto es para evitar recibir mensajes duplicados. También, cuando el nivel de energía del nodo sensor es bajo, el nodo sensor anuncia a todos sus vecinos que le queda poca energía y no puede participar en los mensajes de enrutamiento. La energía restante se reserva para percepción (sensing).

El plano de administración de movilidad detecta y registra el movimiento de los nodos sensores, de esta manera la ruta de retorno hacia el usuario se mantiene siempre, y los nodos sensores pueden no perder de vista quiénes son sus nodos sensores vecinos.

El plano de administración de la tarea equilibra y planifica las tareas de percepción en una región determinada. No todos los nodos sensores en esa región son requeridos para desempeñar la tarea de percepción al mismo tiempo. Como resultado, algunos nodos sensores desempeñan la tarea más que otros dependiendo de su nivel energético.

1.1. El nivel físico

Durante el diseño del nivel físico para las redes de sensores, la minimización del gasto energético asume una importancia significativa, además de la propagación y los efectos del fading.
En general, la mínima potencia de salida necesaria para transmitir un señal a través de una distancia d es proporcional a dn, donde 2 < = n < 4. El exponente n es cercano a 4 para antenas de baja altura y canales cercanos al suelo, como es típico en las comunicaciones de las redes de sensores. Esto puede ser atribuido a la cancelación parcial de la señal debida a la reflexión con el suelo.
Es importante que el diseñador esté enterado de las características intrínsecas de las redes de sensores y las explote al máximo. Por ejemplo, la comunicación multisalto en una red sensora puede efectivamente vencer los efectos del (shadowing) y (path loss), si la densidad de los nodos es suficientemente alta.
Mientras que las pérdidas de propagación y la capacidad de canal limitan la fiabilidad de los datos, este mismo hecho se puede utilizar para la reutilización espacial de la frecuencia.

La elección de un buen esquema de modulación es crítica para una comunicación confiable en una red sensora. Mientras que un esquema M-ario puede reducir el tiempo de transmisión mediante el envío de múltiples bits por símbolo, esto resulta en una circuitería compleja y un creciente consumo energético. Se concluye, por tanto, que bajo condiciones dominantes de ahorro energético, un esquema de modulación binaria es más eficiente energéticamente.

Las tecnologías ultra-wide band (UWB) o impulse radio (IR) han atraído recientemente un considerable interés para su aplicación en comunicaciones, especialmente en redes wireless en interiores.

1.1.1. Ramas de investigación abiertas

El nivel físico es un área bastante inexplorada en las redes de sensores. Las ramas de investigación abiertas abarcan desde diseño de transceptores eficientes energéticamente hasta esquemas de modulación:

- *Esquemas de modulación*: Es necesario desarrollar esquemas de modulación simples y de bajo consumo para las redes de sensores.
- *Estrategias para superar los efectos de propagación de la señal*
- *Diseño de hardware*: Unidades transceptora, sensora y procesadora de tamaño minúsculo, bajo consumo y bajo coste necesitan ser diseñadas. Estrategias de administración del hardware eficientes energéticamente son también esenciales. Algunas estrategias son la gestión de las frecuencias de operación, la reducción del gasto energético on/off, y la predicción de la carga de trabajo en los procesadores.

1.2. El nivel de enlace

El nivel de enlace es responsable de la multiplexación de los flujos de datos, detección de las tramas de datos, acceso al medio y control de errores. Asegura conexiones fiables punto a punto y punto a multipunto en una red de comunicaciones.

1.2.1. Control de acceso al medio

El protocolo MAC en una red de sensores wireless multisalto auto-organizativa debe conseguir dos propósitos.
El primero es la creación de la infraestructura básica de la red necesaria para la comunicación inalámbrica salto a salto y que proporciona a la red sensora la habilidad de auto-organizarse.
El segundo objetivo es compartir eficientemente los recursos de comunicación entre los nodos sensores.

Han sido propuestos mecanismos de acceso al medio basados en asignación fija (fixed allocation) y acceso al azar (random access).
Esquemas MAC basados en demanda pueden ser inapropiados para las redes de sensores debido al gran overhead de sus mensajes así como al retardo de configuración de los enlaces. La conservación energética se consigue mediante el uso de modos de operación de bajo consumo y prefiriendo el vencimiento de temporizadores antes que mensajes de reconocimiento, cuando sea posible.

En la Tabla 1.1 se puede ver una descripción cualitativa de los principales protocolos MAC propuestos. La columna titulada “sensor network specifics” aspira a ilustrar las características nuevas e importantes en cada uno de los diseños que permiten su aplicación en el dominio de la redes de sensores. Presenta las diferencias y desviaciones respecto a los protocolos MAC tradicionales, que por sí mismos no podrían ser aplicables. También se muestra en la columna “power conservation” como cada uno de estos diseños consigue eficiencia energética.

MAC porotocol	Channel acess mode	Sensor network specifics	Power conservation
SMACS and EAR [13]	Fixed allocation ot duplex time slots at fixed frequency	Exploitation of large avaliable bandwidth compared to sensor data rate	Random wake up during setup and tuning radio off while idle
Hybrid TDMA/FDMA [8]	Centralized frequency and time diivision	Optimum number of channels calculated for minimum system energy	Hardware-based approach for system energy minimization
CSMA-based [9]	Contention-based random access	Application phase shift and pretransmit delay	Constant listening time for energy efficiency

Tabla 1.1. Principales protocolos MAC propuestos

1.2.2. Modos de operación con ahorro energético

Para prolongar el tiempo de vida de la red, un nodo sensor debe entrar en periodos de reducida actividad cuando se le esté agotando la batería.

El modo más obvio es apagar el transceptor cuando éste no se requiere. A pesar de que este método parece proporcionar una ganancia significativa de energía, no debe pasarse por alto el hecho que los nodos sensores se comunican usando paquetes de datos cortos.
Cuanto más pequeños son los paquetes, más energía se requiere para encender el transceptor (debido a que tendremos que hacerlo más veces en X tiempo).
De hecho, si nosotros apagamos ciegamente la radio durante cada slot vacío, al cabo de un periodo de tiempo puede ser que terminemos gastando más energía que si la radio se hubiese dejado encendida.

En conclusión, la operación en un modo de ahorro energético es eficiente energéticamente únicamente si el tiempo de permanencia en este estado es mayor que un cierto umbral.
Puede haber un cierto número de estos modos de operación útiles para un nodo sensor wireless, cada uno de estos modos puede ser caracterizado por su consumo energético y por su “latency overhead”, esto es la energía de transición desde y hacia este modo.
La gestión de la enumeración y la transición de estos modos está en investigación.
Un esquema de administración dinámica de la energía para redes de sensores wireless es discutido en [1], donde son propuestos cinco modos de ahorro energético y se investigan las políticas de transición entre modos.

1.2.3. Control de errores

Los dos modos más importantes de control de errores en redes de comunicaciones son el FEC (forward error correction) y el ARQ (automatic repeat request).
La utilidad del ARQ en redes de sensores multisalto está limitada por el coste energético de las retransmisiones adicionales y por el overhead.
Por otro lado, la complejidad de decodificación es mayor en el FEC dado que necesitan ser implementadas las capacidades de corrección de errores.
Se encuentra que el FEC es generalmente ineficiente si la decodificación es llevada a cabo utilizando un microprocesador, y por tanto se recomienda un decodificador Viterby dedicado en la placa.

Considerando esto, códigos sencillos de control de errores con codificación y decodificación de baja complejidad pueden presentar las mejores soluciones para las redes de sensores.

1.2.4. Campos de investigación abiertos

El diseño de protocolos de la capa de enlace está todavía ampliamente abierto a la investigación.
Algunos de los temas en investigación incluyen:

- *MAC para redes de sensores móviles*
  Los protocolos propuestos SMACS y EAR actúan bien únicamente en redes de sensores principalmente estáticas. Estos algoritmos deben ser mejorados para actuar con una movilidad más extensa de nodos sensores y de blancos. Soluciones de movilidad, detección de portadora, y mecanismos de backoff para el esquema basado en CSMA también permanecen altamente inexplorados.
- *Determinación de límites más bajos de consumo energético requeridos para la auto-organización de la red sensora.*
- *Esquemas de código de control de errores*
- *Modos de operación de ahorro energético*

1.3. El nivel de red

Las técnicas tradicionales de enrutamiento ad-hoc no se ajustan normalmente a los requerimientos de las redes de sensores debido a las razones explicadas anteriormente. La capa de red de las redes de sensores se diseña normalmente de acuerdo con los siguientes principios:

- La eficiencia energética es siempre una consideración importante.
- Las redes de sensores son sobretodo orientadas a datos.
- La agregación de los datos es útil únicamente cuando no impide el esfuerzo colaborativo de los nodos sensores.
- Una red de sensores ideal tiene conocimiento de localización y direccionamiento basados en atributos (cualidades)

Rutas eficientes energéticamente pueden ser encontradas basándose en la potencia disponible (PA) en los nodos o mediante la energía requerida (α) para la transmisión en los enlaces a lo largo de las rutas.

Otro objetivo importante es que el enrutamiento pueda estar basado en el punto de vista de orientación a datos. En el enrutamiento orientado a datos, la difusión del interés se realiza para asignar las tareas de percepción a los nodos sensores. Existen dos modos utilizados para realizar la difusión del interés: el sink envía el interés en broadcast [2], y los nodos sensores envían un broadcast anunciando los datos disponibles [3] y esperan una petición por parte de los nodos interesados.
El enrutamiento orientado a datos requiere de denominación basada en atributos.
En la denominación basada en atributos, los usuarios están más interesados en preguntar un atributo del fenómeno antes que preguntar a un nodo determinado.

Por ejemplo, “las áreas donde la temperatura sea superior a 70ºF” es una consulta más común que “la temperatura leída por un nodo determinado”.
La denominación basada en atributos se utiliza para llevar a cabo consultas mediante el uso de los atributos del fenómeno.

La agregación de datos es una técnica utilizada para solventar los problemas de implosión y cubrimiento en enrutamiento orientado a datos [3]. En esta técnica, una red de sensores es normalmente percibida como un árbol de multicast inverso, donde el sink solicita a los nodos sensores que informen sobre las condiciones ambientales del fenómeno. Los datos provenientes de múltiples nodos sensores son agregados si tratan sobre un mismo atributo del fenómeno cuando alcanzan un mismo nodo enrutador en el camino de vuelta.

También, debe tenerse mucho cuidado en la operación de agregar datos, porque las especificaciones de los datos (por ejemplo, las localizaciones desde donde informan los nodos sensores) podrían no ser desechables. Estas especificaciones pueden ser necesarias para ciertas aplicaciones.
Otra importante función del nivel de red es proporcionar interconexión con redes externas como bien otras redes de sensores, sistemas de control y de comandos, y Internet. En un escenario, los nodos sink pueden ser utilizados como una puerta de enlace (gateway) hacia otras redes. Otro escenario consiste en crear un backbone mediante la interconexión de los nodos sink y proporcionar a este backbone acceso a otras redes mediante una puerta de enlace (gateway).

1.3.1. Vías de investigación abiertas

Un repaso de los protocolos propuestos para las redes de sensores puede verse en la Tabla 2.2. Estos protocolos necesitan ser mejorados o desarrollar nuevos protocolos para direccional mayores cambios de topología y mayor escalabilidad.

Protocolos propuestos	Descripción
SMECN [18]	Creates a subgraph of the sensor network that constains the minimum energy path
Flooding	Broadcast data to all neighbor nodes regardless if they receive it before or not
Gossiping [19]	Sends data to one randomly selected neighbor
SPIN [15]	Sends data to sensor nodes only if they are interested; has three typres of mesasages (i.e., ADV, REQ, and DATA)
SAR [13]	Creates multipe trees where the root of each ttree is one hop neighbor from the sink; selects a tree for data to be routed back to the sink according to the energery resources and additive QoS metric
LEACH [16]	Forms clusters to minimize energy dissipation
Directed diffusion [5]	Sets up gradients for data to flow from the source to sink during interest dissemination

Tabla 2.2. Principales protocolos propuestos de nivel de red
(En la bibliografía, [5] es [2] y [15] es [3])

1.4. Nivel de transporte

Este nivel es especialmente necesario cuando se planea que el sistema sea accesible a través de Internet u otras redes externas.
TCP con su actual mecanismo de ventana de transmisión no se adapta a las características extremas que requiere el entorno de una red sensora.
Un mecanismo como TCP splitting puede ser necesario para hacer las redes de sensores interactuables con otras redes como Internet. En este mecanismo, las conexiones TCP son terminadas en los nodos sink, y un protocolo especial de capa de transporte puede manejar las comunicaciones entre el nodo sink y los nodos sensores.
Como resultado, la comunicación entre el usuario y el nodo sink es mediante UDP o TCP vía Internet o vía satélite; por otro lado, la comunicación entre el sink y los nodos sensores puede ser puramente mediante protocolos del tipo de UDP, debido que cada nodo sensor tiene memoria limitada.
Una gran diferencia es que los mecanismos de comunicación extremo a extremo en redes de sensores no están basados en direccionamiento global, sino en direccionamiento basado en atributos. El direccionamiento basado en atributos se explica en una sección anterior.
Factores como el consumo energético y la escalabilidad, y características como el enrutamiento orientado a datos significan que las redes de sensores necesitan manejarse diferente en el nivel de transporte.
Estos requerimientos acentúan la necesidad de nuevos tipos de protocolos de nivel de transporte.

1.4.1. Vías de investigación abiertas

El desarrollo de protocolos de nivel de transporte es un reto desafiante especialmente por las limitaciones hardware, de recursos energéticos y de memoria. Debido a esto, cada nodo sensor no puede almacenar grandes cantidades de datos como un servidor de Internet, y los reconocimientos son muy costosos para las redes de sensores.

1.5. Nivel de aplicación

A pesar de que han sido definidas y propuestas muchas áreas de aplicación, los protocolos de aplicación potenciales siguen siendo una región en gran parte inexplorada.

Vamos a examinar tres posibles protocolos del nivel de aplicación, actualmente en vías de investigación.

1.5.1. Sensor Management Protocol

El diseño de un protocolo de gestión en el nivel de aplicación tiene diversas ventajas. Un protocolo de gestión en el nivel de aplicación hace el hardware y software de las capas inferiores transparente a las aplicaciones de gestión de las redes de sensores.

1.5.2. Task Assignment and Data Advertisement Protocol

Un protocolo de nivel de aplicación que proporcione al usuario software con interfaces eficientes para la diseminación del interés.
Los usuarios envían sus intereses a un nodo sensor, a un subconjunto de nodos o a la red entera. Este interés puede ser sobre un determinado atributo del fenómeno o el accionamiento de un evento. Otro acercamiento es el anuncio de datos disponibles en el cual los nodos sensores advierten los datos disponibles a los usuarios, y los usuarios piden la información en la que están interesados.

1.5.3. Sensor Query and Data Dissemination Protocol

SQDDP proporciona al usuario aplicaciones con interfaces para realizar consultas, responder a consultas y recolectar las respuestas que llegan.
Las consultas no están generalmente dirigidas a unos nodos en particular, sino que suelen basarse en atributos o localización.
Por ejemplo, “las localizaciones de los nodos que perciban temperaturas superiores a 70ºF” es una consulta basada en atributos. Similarmente, “las temperaturas leídas por los nodos en la región A” es un ejemplo de denominación basada en localización

ANEXO 2: CÓDIGOS DE LAS APLICACIONES NESC

2.1. Aplicación generadora de tono

2.1.1. Archivo de configuración del componente

```
// $Id: GeneraTono.nc,v 1.3 2004/10/11 13:30 $

/*
 * Author/Contact:      ignacioBorraz@gmail.com
 *
 * Description:
 *
 * GeneraTono is an application used for improve the sounder
 * functionality.
 * It periodically generates a 4,3 Khz tone using the buzzer of the mica
 * sensor board, and displays the red led toggling every time.
*/

configuration GeneraTono {

// this module does not provide any interface

}

implementation

{

  components Main, GeneraTonoM, LedsC, TimerC, Sounder, IntToRfm;

  Main.StdControl -> TimerC;
  Main.StdControl -> GeneraTonoM;
  Main.StdControl -> IntToRfm;

  GeneraTonoM.SounderControl -> Sounder;
  GeneraTonoM.Leds -> LedsC;
  GeneraTonoM.Timer -> TimerC.Timer[unique("Timer")];
  GeneraTonoM.Timer2 -> TimerC.Timer[unique("Timer")];
  GeneraTonoM.IntOutput -> IntToRfm;

}
```

2.1.2. Archivo de módulo del componente

```
// $Id: GeneraTonoM.nc,v 1.3 2004/17/11 18:20 $

/*
 * Authors:       Ignacio Borraz
 *                Telematic Research UPC Lab
 *
 * Modified:      Modification of the code so that the interval of rest
 *                and the one of tone emission are different.
 *                With this modification tone is sent during 1 second,
 *                and the time of rest is of 9 seconds
 */

module GeneraTonoM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Timer as Timer2;
    interface StdControl as SounderControl;
    interface Leds;
    interface IntOutput;
  }
}

implementation {

// declaration of the static variables of the module

      bool estado;
      int count;

// implementation of StdControl interface

      command result_t StdControl.init()
      {
            estado = FALSE;
            call Leds.init();
            call SounderControl.init();
            return SUCCESS;
      }

      command result_t StdControl.start()
      {
            return call Timer.start(TIMER_REPEAT, 5000);
      }

      command result_t StdControl.stop()
      {
            return call Timer.stop();
      }
```

```
// implementation of Timer.fired event tied code

      event result_t Timer.fired()
      {
            estado = !estado;
            if(estado)
            {
                  count++;
                  return call IntOutput.output(count);
            }
      }

// implementation of message send tied code

      event result_t IntOutput.outputComplete(result_t success)
      {
            if(success == 0) count --;

            call Leds.redToggle();
            call SounderControl.start();
            call Timer2.start(TIMER_ONE_SHOT, 1024);

            return SUCCESS;
      }

// implementation of Timer2.fired event tied code

      event result_t Timer2.fired()
      {
            call Leds.redToggle();
            call SounderControl.stop();
            return SUCCESS;
      }

}
```

2.2. Aplicación receptora de tono

2.2.1. Archivo de configuración del componente

```
// $Id: SendReceiveSystime.nc,v 1.3 2004/19/11 10:50 idgay Exp $
/*
 * Authors:  Ignacio Borraz
 *
 * Description:
 *
 * SendReceiveSystime is an application used to demonstrate the tone
 * detector capabilities and to calculate the sound signal time-of-flight
 * in clock ticks.
 * It toggles the RED led every time that receives and detect a tone
 * generated by the buzzer of the mica sensor board and toggles the
 * YELLOW every time that receives a radio message.
*/

includes IntMsgBB;
includes IntMsg;

configuration SendReceiveSystime {

// this module does not provide any interface

}

implementation

{

      components Main, SendReceiveSystimeM, LedsC, MicCBis, SysTimeM,
      GenericComm as CommBase;

      Main.StdControl -> SendReceiveSystimeM.StdControl;
      Main.StdControl -> CommBase;
      Main.StdControl -> SysTimeM;

      SendReceiveSystimeM.Leds -> LedsC;
      SendReceiveSystimeM.MicControl -> MicCBis;
      SendReceiveSystimeM.Mic -> MicCBis;
      SendReceiveSystimeM.MicInterrupt -> MicCBis;
      SendReceiveSystimeM.Temporizador -> SysTimeM;
      SendReceiveSystimeM.Send -> CommBase.SendMsg[AM_INTMSGBB];
      SendReceiveSystimeM.ReceiveInt -> CommBase.ReceiveMsg[AM_INTMSG];
}
```

2.2.2. Archivo de módulo del componente

```
// $Id: SendReceiveSystimeM.nc,v 1.0 2004/24/11 10:44 idgay Exp $
/*
 * Authors:       Ignacio Borraz
 *                Telematic Research UPC lab
 *
 * Description:
 *
 * View configuration file SendReceiveSystime
*/

includes IntMsgBB;
includes IntMsg;

module SendReceiveSystimeM
{
      provides {
            interface StdControl;
      }
      uses {
            interface Leds;
            interface StdControl as MicControl;
            interface Mic;
            interface MicInterrupt;
            interface SysTime as Temporizador;
            interface SendMsg as Send;
            interface ReceiveMsg as ReceiveInt;
      }
}

implementation {

// declaration of the static variables of the module

   int count;
   uint32_t ticks;
   uint32_t ticksTone;
   uint32_t ticksMsg;
   struct TOS_Msg datos;
   IntMsgBB *message = (IntMsgBB *)datos.data;

// implementation of StdControl interface

      command result_t StdControl.init()
      {
            call Leds.init();
            call MicControl.init();
            call Mic.gainAdjust(64);
            return SUCCESS;
      }
      command result_t StdControl.start()
      {
            call MicControl.start();
            return SUCCESS;
      }
```

```
      command result_t StdControl.stop()
      {
            call MicControl.stop();
      }

 // implementation of tone detected event tied code

      async event result_t MicInterrupt.toneDetected()
      {
            if (count == 0)
            {
                  atomic
                  {
                        ticksTone = call Temporizador.getTime32();
                        ticks = ticksTone - ticksMsg;

                        message->val = ticks;
                        message->src = TOS_LOCAL_ADDRESS;
                  }

                  call     Send.send(TOS_BCAST_ADDR,     sizeof(IntMsgBB),
                  &datos);
            }

            count++;
            call Leds.redToggle();
            call MicInterrupt.enable();

            return SUCCESS;

      }

// implementation of radio message receive event tied code

      event TOS_MsgPtr ReceiveInt.receive(TOS_MsgPtr m)
      {
            atomic
            {
                  ticksMsg = call Temporizador.getTime32();
            }
            count = 0;
            call Leds.yellowToggle();
            return m;

      }

// implementation of correct message send event
      event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
      {
            return SUCCESS;
      }

}
```

ANEXO 3: CÓDIGOS DE LA APLICACIÓN JAVA

3.1. Clase MyListen

```
// $Id: Listen.java,v 1.4 2003/10/07 21:46:08 idgay Exp $

// Modificación de la aplicación Listen que incorpora el sistema
// operativo TinyOS.
// Se ha explicado el funcionamiento de Listen y se ha añadido el código
// necesario para que cumpla las funcionalidades deseadas.

// $Id: MyListen.java,v 1.0 2005/02/14 0:59:23 iborraz Exp $

package net.tinyos.MyListener;

import java.io.*;
// Importación de paquetes propios del sistema operativo TinyOS
import net.tinyos.packet.*;
import net.tinyos.util.*;
import net.tinyos.message.*;

public class MyListen {

    public static void main(String args[]) throws IOException {

      int valor, mote1, mote2, mote3;
      valor = mote1 = mote2 = mote3 = 0;
      // Definimos una variable valor que será una variable intermedia
      // hasta que se compruebe que mote ha generado la medida y se le
      // asigne a su ID correspondiente.

      boolean mote1B, mote2B, mote3B;
      mote1B = mote2B = mote3B = false;

      if (args.length > 0) {
          System.err.println("usage: java net.tinyos.tools.Listen");
          System.exit(2);
      }
      // Este programa funciona sin argumentos, por ello se añade un
      // control de errores para evitar que sea llamado de forma
      // indebida.

      PacketSource reader = BuildSource.makePacketSource();

      // Se define un objeto "reader" del tipo proporcionado por la
      // interfaz PacketSource ubicada en net.tinyos.packet. Para crearlo
      // se llama al método "makePacketSource()" de la clase
      // "BuildSource" ubicada también en net.tinyos.packet.
      // Este método captura todos los parámetros necesarios para definir
      // la fuente de paquetes (traducción de PacketSource) de la
      // variable de entorno MOTECOM. Si se investiga el código de la
      // clase Env, se puede observar como utiliza código nativo para
      // acceder a la variable de entorno MOTECOM cargando la DLL
      // (librería) correspondiente "getenv"
```

```
if (reader == null) {
      System.err.println("Invalid packet source (check your MOTECOM
      environment variable)");
      System.exit(2);
}

// Si la instanciación de la fuente de paquetes que nos proporciona
// el objeto "reader" fracasa y la instancia apunta a "null", se
// gestiona el error mostrando un mensaje por pantalla al usuario
// para que chequee que está correctamente definida la variable de
// entorno MOTECOM.
// Como se avisa en BuildSource, también debe comprobarse que se
// encuentre correctamente instalado el código nativo (JNI) de
// net.tinyos.util.Env que permite acceder a las variables de
// entorno

myTableBest frame = new myTableBest();
frame.pack();
frame.setVisible(true);

// Iniciamos la interfaz gráfica creando un objeto de la clase
// myTableBest y haciéndolo visible.

try {
      reader.open(PrintStreamMessenger.err);

      // En la definición del método "open" se permite indicar
      // mediante un parámetro el destino de mensajes informativos
      // que provengan de la fuente. En este caso, estos mensajes
      // los captura un objeto del tipo PrintStreamMessenger que
      // se crea cuando llamámos al método PrintStreamMessenger.err
      // que se encuentra definido en la clase PrintStreamMessenger
      // en net.tinyos.util.

  for (;;) {
      byte[] packet = reader.readPacket();

      // Permanecemos siempre a la escucha de paquetes que
      // provengan de la fuente. Cuando llega un paquete, es
      // almacenado en un objeto del tipo array de bytes.


      // Los paquetes que provienen de los motes, sabemos que
      // tienen una estructura común, que se nos explica en la
      // clase PacketSource:
      //
      // "The packet byte array must have the following format:
      //          - a TinyOS TOS_Msg header (5 bytes):
      //                address: 2 bytes, little endian
      //                AM type: 1 byte
      //                group: 1 byte
      //                length: 1 byte
      //          - 'length' data bytes
      //
      // Nosotros hemos definido que los paquetes radio generados
      // por el mote emisor del tono, tienen 4 bytes de datos
```

```
    // (2 bytes para el contador, y otros 2 bytes para la ID
    // asignada), por tanto miden en total 9 bytes.
    // Por contra, hemos definido que los paquetes radio
    // generados por el mote receptor y que contienen la medida
    // en ticks de reloj que nos interesa, tienen 6 bytes de
    // datos: 4 bytes del temporizador (para asegurar que no da
    // la vuelta) y los 2 bytes de la ID asignada. Por tanto
    // miden 11 bytes.

  if(packet.length == 11)
  {
    valor = CapturaDatos.capturaMedida(packet);

    // Si el paquete pertenece a uno de los motes receptores
    // llamamos a un método encargado de capturar el valor de los
    // ticks de reloj del temporizador.
    // Para aumentar la modularidad y clarificar el código, se ha
    // decidido crear una clase aparte llamado "CapturaDatos" que
    // implementará este método.
    // A su finalización, almacenamos en la variable int "valor"
    // el número de ticks ya transformado a decimal que contenía
    // el paquete recibido.

    if((CapturaDatos.capturaGenerica(packet,10)== 1) & (mote1B ==
    false))
    {
          mote1 = valor;
          mote1B = true;
    }
    if((CapturaDatos.capturaGenerica(packet,10)== 2) & (mote2B ==
    false))
    {
          mote2 = valor;
          mote2B = true;
    }
    if((CapturaDatos.capturaGenerica(packet,10)== 3) & (mote3B ==
    false))
    {
          mote3 = valor;
          mote3B = true;
    }
    // vamos a considerar que en nuestro escenario tenemos un
    // mote emisor de tono (de quien se desconoce la posición) y
    // tres motes detectores de tono (que toman medida de
    // distancia y la envían al pc).
    // Si recordamos los dos últimos bytes de datos, bytes 10 y
    // 11 en este tipo de mensajes, contienen la ID del
    // dispositivo. Esta ID puede ser asignada en el momento de
    // subir el código al mote y en este caso se supone que se
    // han asignado las ID 1, 2 y 3 a los tres motes detectores
    // de tono.
    // Mediante esta sucesión de "if" se evalua a que mote
    // pertenece la medida recibida y si todavía no ha llegado
    // ninguna de él (contemplamos la posibilidad de recibir 2 o
    // más medidas de un mismo mote antes de recibir la primera
    // medida de otro de ellos y en este caso descartamos esa
    // segunda medida) se asigna la medida a otra variable int.
```

```
            if(mote1B == mote2B == mote3B == true)
            {
                  String posicion = MathLocalization.Triangulation(mote1,
                  mote2, mote3);

                  // Cuando ya hemos recibido tres medidas válidas, una
                  // perteneciente a cada uno de nuestros motes
                  // detectores de tono, ya podemos llamar a la clase
                  // matemática MathLocalization que nos proporciona el
                  // método  "Triangulation" para realizar la
                  // triangulación y encontrar la posición del mote que
                  // buscamos.

                  frame.AsignarValor(mote1, mote2, mote3, posicion);

                  // Una vez disponemos de la posicion, llamamos al
                  // método AsignarValor definido en nuestra
                  // implementación de la interficie gráfica que muestra
                  // los resultados en una tabla, para que actualize los
                  // campos de la tabla y se visualize la posicion
                  // calculada así como los ticks que han dado cada uno
                  // de los nodos de referencia.

                  mote1B = mote2B = mote3B = false;

                  // Volvemos a cambiar los valores de los indicadores
                  // boolean a "false" para que vuelva a repetirse el
                  // proceso de coger valores provenientes de los
                  // diferentes motes y computar una nueva posición para
                  // el nodo ilocalizado.
            }

            }
            // Cierre del if(packet.length == 11)

            Dump.printPacket(System.out, packet);
            System.out.println();

            // Con el paquete capturado en un array de bytes, llamámos al
            // método  "printPacket" de la clase "Dump" perteneciente al
            // paquete net.tinyos.message que se encarga de imprimir por
            // pantalla el valor en hexadecimal de cada uno de los bytes
            // del paquete y se queda a la espera de recibir un nuevo
            // paquete de la fuente.
      } // cierre del for(;;)
      }
      catch (IOException e) {
          System.err.println("Error on " + reader.getName() + ": " + e);
      }

      // Encontramos la gestión de excepciones try-catch debido a que en
      // la interfaz PacketSource donde se define el método "open", se
      // define que éste podría lanzar una excepción del tipo IOException
      // si la fuente no puede abrirse
    }
}
```

3.2. Clase CapturaDatos

```
package net.tinyos.MyListener;

import java.io.*;

public class CapturaDatos {

	// Clase encargada de capturar los ticks de reloj del temporizador
	// y pasarlos a su valor decimal.

	public static int capturaMedida ( byte[] packet)
	{
		String Str5, Str6, Str7, Str8, Concat;
		int valor = 0;

		// Definimos 4 cadenas de carácteres del tipo String, cada
		// una de ellas contendrá el valor hexadecimal del byte que
		// su propio número indica.
		// Los bytes que contienen la información de ticks en el
		// paquete son los bytes 6, 7, 8 y 9, los índices que se
		// corresponden a estos bytes son [5], [6], [7] y [8]

		Str5 = HexByte(packet[5]);
		Str6 = HexByte(packet[6]);
		Str7 = HexByte(packet[7]);
		Str8 = HexByte(packet[8]);

		// Asignamos el String que devolverá el método HexByte,
		// encargado de transformar el valor contenido en un byte a
		// hexadecimal al Str que corresponde en cada caso.

		Concat = Str8 + Str7 + Str6 + Str5;

		// ATENCIÓN: El orden en que colocamos los valores es inverso
		// al orden de entrada. Esto se debe a que los campos del
		// paquete origen se transfieren en little endian, es decir,
		// de byte menos significativo a byte más significativo.
		// Dado que ahora queremos realizar la conversión a decimal,
		// necesitamos que el valor total hexadecimal esté en orden
		// contrario, en big endian.

		valor = Integer.parseInt( Concat,16 );
		return valor;

		// Parseamos a entero decimal int el valor del String Concat
		// especificando en que base se encontraba el número que
		// contenía, en este caso, siendo hexadecimal, base 16.
	}

	public static int capturaGenerica ( byte[] packet, int num)
	{
		// Definimos un método de captura genérica para transformar a
		// int, cualquier posición del array de bytes que forma el
		// paquete.
```

```
            return Integer.parseInt( HexByte(packet[num-1]), 16);

            // la condición es "packet[num+1]" para que el hecho de que
            // este valor sirva como indice del array sea transparente y
            // en la aplicación que llame a esta función se tenga que
            // pensar únicamente la posición real, en este caso la décima
            // posición del array.
      }

      public static String HexByte(int b) {

      // Método de conversión de byte a hexadecimal (realmente nos
      // valemos de las conversiones automáticas de Java.
      // La conversión de tipo byte a tipo int cumple los dos requisitos
      // básicos:
      //          - Los dos tipos son compatibles
      //          - El tipo de destino es más grande que el tipo fuente

      String bs = Integer.toHexString(b & 0xff).toUpperCase();
      if (b >=0 && b < 16)
          return ("0" + bs);
      return (bs);
    }
}
```

3.3. Clase MathLocalization

```
package net.tinyos.MyListener;

import java.io.*;
import java.lang.*;
import java.lang.Math.*;

public class MathLocalization {

      // Clase encargada de realizar todas las operaciones matemáticas
      // directamente vinculadas con la tarea de localización.
      // Sus dos cometidos principales serán la conversión a distancias
      // de los valores de ticks de temporizador, y el algoritmo de
      // triangulación en sí mismo.

      public static double CalculaDistancia ( int tick)
      {
      // Función encargada de recibir el valor en ticks de reloj y
      // convertirlo a metros

            double conversion = 1085E-9;

            // COMENTARIO: es la precisión de nuestro reloj, 1/921600

            int meters = 343;
            double dist = (tick*conversion)*meters;
            return dist;
      }
```

```
public static double DistanciaReferencia (double X1, double Y1,
double X2, double Y2)
{
      // Este método se encarga de, dadas las coordenadas de dos
      // puntos, calcular la distancia que los separa

      double DR = (X2-X1) - (Y2-Y1);
      return Math.sqrt(DR);
}



public static String Triangulation ( int mote1, int mote2, int
mote3)
{

      double X_POS_MOTE1 = 0;
      double X_POS_MOTE2 = 1;
      double X_POS_MOTE3 = 2;
      double Y_POS_MOTE1 = 0;
      double Y_POS_MOTE2 = 1;
      double Y_POS_MOTE3 = 0;

      // Los motes 1, 2 y 3 son motes de referencia, es decir,
      // la aplicación del PC, conoce sus posición sobre un eje de
      // cordenadas ficticio.
      // Hemos basado este eje en la posición del mote 1, es decir
      // este mote es el origen y por tanto tiene coordenadas (0.0)

      double     dist1_2     =     DistanciaReferencia(X_POS_MOTE1,
      Y_POS_MOTE1, X_POS_MOTE2, Y_POS_MOTE2);
      double     dist1_3     =     DistanciaReferencia(X_POS_MOTE1,
      Y_POS_MOTE1, X_POS_MOTE3, Y_POS_MOTE3);
      double     dist2_3     =     DistanciaReferencia(X_POS_MOTE2,
      Y_POS_MOTE2, X_POS_MOTE3, Y_POS_MOTE3);

      // Para poder realizar los cálculos que implica el algoritmo
      // de triangulación necesitamos conocer las distancias de los
      // nodos de referencia entre sí mismos.


      double mote1Dist, mote2Dist, mote3Dist;
      String distancia = new String();
      mote1Dist = CalculaDistancia (mote1);
      mote2Dist = CalculaDistancia (mote2);
      mote3Dist = CalculaDistancia (mote3);

      // TRIANGULACION

      double alfa;
      double beta;

      // Los ángulos alfa y beta son necesarios para la obtención
      // de la posición del nuevo nodo en nuestros ejes de
      // coordenadas.
```

```
        // Para encontrar estos dos ángulos se crean dos triángulos
        // ficticios. El primero entre el mote 1 (nodo origen), mote
        // 3 y nuestro mote ilocalizado. El segundo entre el mote 1
        // (nodo origen), mote 2 y nuestro mote ilocalizado

        double exp = 2;
        double angulo_inicial;

        //Para la determinación de la posición en el eje de las "y"
        // del nuevo nodo es necesario verificar la condición
        // beta = |alfa - angulo_inicial |.
        // Hemos denominado angulo_inicial al ángulo formado entre el
        // mote 1 y el mote 2.

        angulo_inicial = Math.acos(dist1_3/dist1_2);

        alfa     =     Math.acos(     (Math.pow(mote1Dist,exp)     +
        Math.pow(dist1_3,exp)     -     Math.pow(mote3Dist,exp))     /
        (2*mote1Dist*dist1_3));

        beta     =     Math.acos(     (Math.pow(mote1Dist,exp)     +
        Math.pow(dist1_2,exp)     -     Math.pow(mote2Dist,exp))     /
        (2*mote1Dist*dist1_2));

        // IMPORTANTE: Los métodos trigonométricos del paquete Math
        // devuelven los ángulos en radianes, no en grados.

        double X_POS_MOTENUEVO;
        double Y_POS_MOTENUEVO;

        X_POS_MOTENUEVO = mote1Dist*Math.cos(alfa);

        if( beta == Math.abs(alfa-angulo_inicial))
              Y_POS_MOTENUEVO = mote1Dist*Math.sin(alfa);
        else
              Y_POS_MOTENUEVO = -mote1Dist*Math.sin(alfa);

        // El fragmento de código superior, calcula mediante
        // trigonometría la posición x del nuevo nodo y la posición
        // y, verificando la condición que se ha expresado antes.

        distancia = ("(" + X_POS_MOTENUEVO + "," + Y_POS_MOTENUEVO +
        ")");

        // Almacenamos la posición (x,y) en una variable
        // String y se la devolvemos al main.

        return distancia;

        // el método de triangulación devuelve un String donde se
        // encuentra el resultado de la triangulación, es decir, una
        // posición absoluta calculada en función de las distancias
        // del nodo respecto a los nodos de referencia con posición
        // conocida.
    }
```

3.4. Clase MyTableBest

```
package net.tinyos.MyListener;

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.text.*;

public class myTableBest extends JFrame {
      private int j = 0;
      protected ModeloTabla miModelo;
      protected JTable Tabla;

      // Esta clase va a ser la encargada de dibujar y crear el Frame
      // donde vamos a crear una tabla del tipo que hemos definido
      // nosotros mismos en "ModeloTabla"

      public myTableBest()
      {
            super("Java Localization System");

            try {

                  UIManager.setLookAndFeel("com.sun.java.swing.plaf.windo
                  ws.WindowsLookAndFeel");
            }
            catch (Exception e) {
                  JOptionPane.showMessageDialog(null,"Error  al  intentar
                  cargar L&F");
            }

            miModelo = new ModeloTabla();
            Tabla = new JTable(miModelo);

            Tabla.setPreferredScrollableViewportSize(new    Dimension(900,
            300));

            JScrollPane scrollPane = new JScrollPane(Tabla);
            getContentPane().add(scrollPane, BorderLayout.CENTER);

            menus ();

            JPanel panel= new JPanel();
            JPanel panel1= new JPanel();
            JPanel panel2= new JPanel();
            panel.setBackground(Color.lightGray);
            panel1.setBackground(Color.lightGray);
            panel2.setBackground(Color.lightGray);
            getContentPane().add(panel,BorderLayout.SOUTH);
            getContentPane().add(panel1,BorderLayout.WEST);
            getContentPane().add(panel2,BorderLayout.EAST);
```

```
        setSize(800,580);
        setVisible(true);
        show ();

    }

    public int AsignarValor(double mote1Dist, double mote2Dist, double
    mote3Dist,String posicion)
    {
        Date Hora = new Date();
        SimpleDateFormat sdf = new SimpleDateFormat("dd MMM yyyy
        hh:mm:ss zzz");

        // Como se explica en la clase "ModeloTabla", la quinta
        // posición de nuestra tabla queremos que la ocupe la fecha
        // de proceso de la posición del nodo ilocalizado.
        // La clase Date es la encargada de capturar la hora actual,
        // sin embargo si intentamos mostrar su contenido, nos
        // encontramos que la muestra como el número de milisegundos
        // transcurridos desde el 1 de enero de 1970.
        // Para obtener una forma más amigable de cara al usuario
        // final necesitamos formatear este dato para mostrarlo de
        // forma más convencional.
        // Para ello utilizamos la clase SimpleDateFormat, una
        // subclase predefinida de DateFormat. Creamos un nuevo
        // objeto de esta clase mediante un constructor donde
        // se especifica el formato de salida que queremos para la
        // fecha siguiendo una nomenclatura que nos proporciona la
        // propia clase.


        Object[] linea = {new Double(mote1Dist),new
        Double(mote2Dist), new Double(mote3Dist),posicion,
        sdf.format(Hora)};

        // Creamos un array de objetos con las 3 medidas de ticks,
        // una de cada nodo, el String que contiene la posicion
        // estimada del nodo ilocalizado, y por último utilizamos el
        // método sdf.format para formatear según lo que antes hemos
        // especificado, la hora que nos proporciona el objeto Hora
        // de la clase Date.

        for(int i=0; i<5; i++)
              miModelo.setValueAt(linea[i],j,i);

              // mediante sucesivas llamadas al método "setValueAt"
              // rellenamos todas las columnas de la primera fila.

        if( j == 19) j = 0;
        else j++;

        // Si no añadiesemos esta variable int de control, cada vez
        // que se generase una nueva medida proporcionada por
        // Triangulation, los datos se sobreescribirían en su
        // presentación en la tabla, ya que siempre estaríamos
        // accediendo a la primera fila.
```

```
		// Como el comportamiento que deseamos no es este, mediante
		// el incremento de "j" nos aseguramos de que cada nueva
		// medida ocupará la fila inmediatamente inferior a la que
		// ocupó la anterior medida.
		// Sin embargo, hemos definido en nuestro modelo de tabla
		// ModeloTabla, que nuestra tabla tendría 20 filas, así que
		// nos daría un error de ejecución si sobrepasásemos ese
		// índice cuando rellenamos la tabla.
		// Para evitarlo, cada vez que "j" llega a 19 (fijémonos que
		// evaluamos DESPUÉS de escribir el dato en la tabla, por
		// tanto cuando "j" vale "19" acabamos de rellenar la fila
		// con índice [19] que corresponde a la veiteava fila de la
		// tabla) devolvemos su valor a 0.
		// De este modo la siguiente medida sobreescribirá la medida
		// más antigua que poseíamos en la tabla y podremos darnos
		// cuenta de este hecho gracias a que hemos añadido la
		// visualización de la hora de proceso de cada medida.

		return j;
	}


	public void menus () {
		JMenuBar menus = new JMenuBar();
		JMenu archivo= new JMenu("Archivo");
		JMenuItem nue= new JMenuItem("Nuevo");
		nue.addActionListener (
			new ActionListener () {
				public void actionPerformed (ActionEvent e) {
					nuevo();
				}
			}
		);

		JMenuItem sal= new JMenuItem("Salir");
		sal.addActionListener(
			new ActionListener () {
				public void actionPerformed (ActionEvent e) {
					System.exit(0);
				}
			}
		);

		JMenuItem abr= new JMenuItem("Abrir");
		abr.addActionListener(
			new ActionListener () {
				public void actionPerformed (ActionEvent e) {
				}
			}
		);
		JMenuItem guar= new JMenuItem("Guardar");
		guar.addActionListener (
			new ActionListener () {
				public void actionPerformed (ActionEvent e) {
				}
			}
		);
```

```
            JMenu editar= new JMenu("Editar");
            JMenuItem cor= new JMenuItem("Cortar");
            cor.addActionListener (
                  new ActionListener () {
                        public void actionPerformed (ActionEvent e) {
                        }
                  }
            );
            JMenuItem cop= new JMenuItem("Copiar");
            cop.addActionListener (
                  new ActionListener () {
                        public void actionPerformed (ActionEvent e) {
                        }
                  }
            );
            JMenuItem peg= new JMenuItem ("Pegar");
            peg.addActionListener (
                  new ActionListener () {
                        public void actionPerformed (ActionEvent e) {
                        }
                  }
            );

            JMenu about= new JMenu("Ayuda");
            JMenuItem ayu= new JMenuItem("Ayuda");
            ayu.addActionListener (
                  new ActionListener () {
                        public void actionPerformed (ActionEvent e) {
                              ayuda ();
                        }
                  }
            );

            archivo.add(nue);
            archivo.add(abr);
            archivo.add(guar);
            archivo.addSeparator();
            archivo.add(sal);
            editar.add(cor);
            editar.add(cop);
            editar.add(peg);
            about.add(ayu);
            menus.add(archivo);
            menus.add(editar);
            menus.add(about);
            setJMenuBar(menus);

      }

      public void nuevo () {
            myTableBest frame = new myTableBest();
        frame.pack();
        frame.setVisible(true);

      }
}
```

3.5. Clase ModeloTabla

```
package net.tinyos.MyListener;

import javax.swing.JTable;
import javax.swing.table.AbstractTableModel;
import javax.swing.JScrollPane;
import javax.swing.JFrame;
import javax.swing.SwingUtilities;
import javax.swing.JOptionPane;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.text.*;

public class ModeloTabla extends AbstractTableModel
{

      final String[] NombreColumnas = {"medida mote1",
                                       "medida mote2",
                                       "medida mote3",
                                       "posición estimada",
                                       "hora de cálculo"};

        Object[][] data = {
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
{new Double(0), new Double(0), new Double(0), " ", new StringBuffer()},
        };

      // Lo más parecido a unos valores vacíos conseguido es esto.
      // El constructor de los objectos Double requiere de que se le pase
      // un valor por tanto le pasamos un "0". En el cuarto campo irá un
      // String, así que aquí si que podemos ocupar la posición con un
      // espacio en blanco.
      // En el quinto campo se quiere poner la hora y el día en que ha
      // sido tomada la medida. La forma de conseguir esto se explica en
      // el método "AsignarValor" de la clase myTableBest, sin embargo
      // para inicializar el array de Objetos necesitamos saber de que
```

```
    // tipo será el dato que ocupará la posición. El dato que nos
    // proporciona la fecha es de la clase StringBuffer,que nos
    // proporciona un constructor vacío que hacemos servir.

      public int getColumnCount() {
          return NombreColumnas.length;
      }

      public int getRowCount() {
          return data.length;
      }

      public String getColumnName(int col) {
          return NombreColumnas[col];
      }

      public Object getValueAt(int fila, int col) {
          return data[fila][col];
      }

      public Class getColumnClass(int c) {
          return getValueAt(0, c).getClass();
      }

      public boolean isCellEditable(int fila, int col) {
              return false;
      }

      public void setValueAt(Object value, int fila, int col) {

              data[fila][col] = value;
              fireTableCellUpdated(fila, col);

      }
          // Este es el método más relevante para nosotros, ya que es
          // el encargado de modificar los valores de las celdas de la
          // Tabla.
          // A este método se le pasa un Objeto y la posición de la
          // tabla mediante "row" y "col" donde se le tiene que ubicar.
          // Una vez ubicado, el método llama a "fireTableCellUpdated",
          // encargado de avisar a todos los Listeners que el valor de
          // la celda ha sido actualizado.
}
```

3.6. Modificación de MyListen para el escenario

Como se ha dicho en el capítulo de resultados del presente proyecto, para probar el correcto funcionamiento de la aplicación de triangulación java con el escenario del que se disponía se tuvieron que hacer modificaciones.
Aquí pues se presenta el código con el que se comprobó el funcionamiento conjunto con las aplicaciones nesC y en el que sólo se dispone de un valor de ticks.

```
package net.tinyos.MyListenerMod;

import java.io.*;

import net.tinyos.packet.*;
import net.tinyos.util.*;
import net.tinyos.message.*;

public class MyListenMod {

    public static void main(String args[]) throws IOException {

      int valor, mote1;
      valor = mote1 = 0;
      boolean mote1B;
      mote1B = false;

      if (args.length > 0) {
          System.err.println("usage: java net.tinyos.tools.Listen");
          System.exit(2);
      }

      PacketSource reader = BuildSource.makePacketSource();

      if (reader == null) {
            System.err.println("Invalid packet source (check your MOTECOM
            environment variable)");
            System.exit(2);
      }

      myTableBestMod frame = new myTableBestMod();
      frame.pack();
      frame.setVisible(true);

      try {
        reader.open(PrintStreamMessenger.err);

        for (;;) {

            byte[] packet = reader.readPacket();
            if(packet.length == 11)
            {
                  valor = CapturaDatosMod.capturaMedida(packet);
```

```
                if((CapturaDatosMod.capturaGenerica(packet,10)==  1)  &
                (mote1B == false))
                {
                      mote1 = valor;
                      mote1B = true;
                }

                if(mote1B == true)
                {
                      String posicion = "prueba";
                      frame.AsignarValor(mote1, mote2, mote3, posicion);
                }

          }
          Dump.printPacket(System.out, packet);
          System.out.println();
          }
     }
     catch (IOException e) {
          System.err.println("Error on " + reader.getName() + ": " +
          e);
     }
}
}
```

3.7. Modificación de MyListen para comprobar la triangulación

Código de prueba que también se cita en el capítulo de resultados. Consiste en la modificación de MyListen para que se introduzcan por consola 3 bytes como si fuesen los recibidos por COM1.

```
package net.tinyos.MyListener;

import java.io.*;

public class PruebaListenTriangulation {

     public static void main (String args[]) {

          myTableBest frame = new myTableBest();
          frame.pack();
          frame.setVisible(true);

          int i = 0;
          int resultado, resultado2, resultado3;
          resultado = resultado2 = resultado3 = 0;
          int mote1, mote2, mote3;
          mote1 = mote2 = mote3 = 0;
          boolean mote1B, mote2B, mote3B;
          mote1B = mote2B = mote3B = false;
          byte packet[] = new byte[10];
          byte packet2[] = new byte[10];
          byte packet3[] = new byte[10];
```

```
            if (args.length != 30) {
                  System.err.println("Uso incorrecto: Debe especificar un
                  valor a convertir");
                  System.exit(2);
            }
            for(i=0; i < 10; i++)
            {
                  packet[i] = Byte.parseByte(args[i]);
            }
            for(i=0; i < 10; i++)
            {
                  packet2[i] = Byte.parseByte(args[i+10]);
            }
            for(i=0; i < 10; i++)
            {
                  packet3[i] = Byte.parseByte(args[i+20]);
            }
            resultado = CapturaDatos.capturaMedida(packet);
            resultado2 = CapturaDatos.capturaMedida(packet2);
            resultado3 = CapturaDatos.capturaMedida(packet3);

            if((CapturaDatos.capturaGenerica(packet,10)== 1) & (mote1B ==
            false))
            {
                  mote1 = resultado;
                  mote1B = true;
            }
            if((CapturaDatos.capturaGenerica(packet2,10)==  2)  &  (mote2B
            == false))
            {
                  mote2 = resultado2;
                  mote2B = true;
            }
            if((CapturaDatos.capturaGenerica(packet3,10)==  3)  &  (mote3B
            == false))
            {
                  mote3 = resultado3;
                  mote3B = true;
            }
            if(mote1B == mote2B == mote3B == true)
            {
                  String posicion = MathLocalization.Triangulation(mote1,
                  mote2, mote3);
                  frame.AsignarValor(mote1, mote2, mote3, posicion);
                  mote1B = false;
                  mote2B = false;
                  mote3B = false;
            }

      }
}
```

ANEXO 4: PARTICIPACIÓN EN LA LISTA TINYOS-HELP

La comunidad de desarrolladores del sistema operativo TinyOS dispone de dos herramientas para consultar y difundir información; una es el CVS y la otra las listas de correo.
En este anexo se muestra la evolución que nos ha hecho pasar de plantear dudas a resolver dudas en estos meses de proyecto.

[Tinyos-help] Problem with Listen application

Ignacio Borraz ignacioborraz at gmail.com
Sun Nov 14 10:20:35 PST 2004

Hi all,

I'm working with the Crossbow kit, MIB510 and mica2s and I tried to send some information collected by sensorboard of mica2s from one mote to PC, using a mica2 mote as a TOSBase connected to the MIB510 for allow the pass of the messages from RF to UART, as same as do the example application OscilloscopeRF.

When I execute the Listen application, sometimes the program don't print the raw data of each packet received, only shows a line of points.
I need to reboot the computer more than one time for solve it temporally.

Has anybody know how I can solve this problem?

Thanks in advance.

Ijborraz

[Tinyos-help] About Microphone.

HO MIN KWON ho.kwon at asu.edu
Tue Nov 16 17:03:50 PST 2004

Hellow..
I'm trying to record some signals through microphone on the MTS310CA. And I found one function , Mic.gainAdjust(), to control microphone. However, I couldn't find reference about this. I think better sound samples will be taken by using similar funsctions like this. Can you help me find this?

Thanks,
Homin.

Next message: [Tinyos-help] About Microphone.

Ignacio Borraz ignacioborraz at gmail.com
Wed Nov 17 02:15:53 PST 2004

Hello Homin,

the functions to control microphone are defined in the archive /tos/interfaces/Mic.nc, in this same document are some information or reference about what does the function Mic.gainAdjust(), and another functions like Mic.muxSel().

To consult one existing implementation of this functions, you can see the archive MicM.nc and his configuration archive MicC.nc that you can find in /tos/sensorboards/micasb.

I hope this could help you,

Ijborraz

[Tinyos-help] permission denied

Andre Paul andre.paul at mail.mcgill.ca
Tue Nov 23 07:29:14 PST 2004

Hello

I am trying to install the "CntToLedsAndRfm" exercise unto a mica2 mote using a MIB51O programming board. I get the message

Error: Permission denied ->COM1
make[1]: Leaving directory'/opt/tinyos-1.x/apps/cnttoledsandrfm'

Can anyone help..I desperately need some

Andre

Next message: [Tinyos-help] permission denied

Ignacio Borraz ignacioborraz at gmail.com
Tue Nov 23 07:43:04 PST 2004

Hello Andre,

I think that this problem can occur for two main reasons.

One of these, if you are executing at the same time a application that writes, reads, or does any operation with COM1 (for example the application "Listen" that listens for packet's arrival to COM1)

Another one possibility is that you don't have the necessary system permissions for interactuate with COM1 (in this situation the problem must occur with every application that you try to install in a mote).

Hope it helps,
Ijborraz

[Tinyos-help] confusion about Time and Timers

Selcen Isci sisci at ku.edu.tr
Mon Nov 29 17:29:07 PST 2004

I have confusion about the Timers in tinyos applications.
In the TestTinyViz application TimerC was used and it was wired so that every mote had its own timer. Also there is a time field in the TOSMsg packets which shows the time when the packet is received.
In addition there is a component called SimpleTime which I assume shows the current time by the call to Timer.getLow32(). I have questions about these :
1) Timer.getLow32() returns the current time in which unit? miliseconds or any count numbers?
2) By using TimerC.Timer[unique("Timer")] we make every mote have its own timer. If I want to see the time when the packet is received, does recv_packet->time show the time in that mote or does it shows the time in the simulation time?
3) Is there a way to keep a single timer for all the motes? I thought it could be SimpleTime, but I'm not sure of that.
4) Is there a way to see the simulation time? I thought it could be Timer.getLow32().
Since there is no manual for the explanations of the components and its commands and events, I cannot reach any information. Can someone help?

Thanks in advance,

Selcen

Next message: [Tinyos-help] confusion about Time and Timers

Ignacio Borraz ignacioborraz at gmail.com
Wed Dec 1 09:07:28 PST 2004

Hello Selcen,

I haven't found the component SimpleTimer in the code of application TestTinyViz, and I don't know any interface that provides the function Timer.getLow32(), where is it? I'm interested in.
For my experience with other timers I try to answer your questions.

1) All the functions used to return the value of a timer should return the number of clock ticks. The number of ticks that clock does in a second is the same of his frequency (for example a 916,2 Khz frequency clock does 916200 ticks in a second).

2) Doing recv_packet->time the most habitual would be that returns the time in that mote when the packet was transmitted.(In fact, the time will never be exactly because the value in the field "time" of the packet is adjusted before transmission)

3) I think that it's impossible to do it. Every mote has his internal and individual clocks, exists ways to synchronizing the timers of the diferents motes, but I don't know any way to do your objective. (I'm talking in terms of testing motes, I don't know a lot of the simulation environment)

4) As I have said in point 3, I don't know a lot of the simulation environment because I don't work with it.

Hope it helps,
ijborraz

Next message: [Tinyos-help] confusion about Time and Timers

Selcen Isci sisci at ku.edu.tr
Wed Dec 1 20:20:37 PST 2004

Hi Ignacio,
Thanks a lot for your answers to my questions.
Simple Time is defined in tos/system and Time interface provides the functions get(), getHigh32(), getLow32() and getUs(). In my application in TOSSIM I need to measure the packet transfer latency between motes; therefore I thought it could be useful to have a single timer to do that. As you said it doesn't look possible to achieve this. Do you know any other solution to measure the latency?

Selcen

Next message: [Tinyos-help] confusion about Time and Timers

Ignacio Borraz ignacioborraz at gmail.com
Thu Dec 2 03:33:04 PST 2004

Hi Selcen,

I'm working in a project that I have to do something similar as the objective you propose, and I think that one of my solutions can be adapted for your work.
I think that a posible solution for you could be:

- Implement a code in one mote that transmits two packets consecutively.

- Implement a code in another mote (that mote must have a Timer started when the mote initializes) that takes the value of the Timer (with getLow32() for example) when arrives the first packet, and then takes the value of the Timer when arrives the second packet. After, doing

TimerValueSecondPacket - TimerValueFirstPacket,

you will have a more or less the correct value of the time (on ticks of the clock) that delays the second packet since its transmitted to its received.

Probably, it isn't the best solution, but I think that for a firstly approximation it's quite good.

Hope it helps,
ijborraz

[Tinyos-help] CntToLedsAndRfm ain't working today

Wong Yee Kun ykwong at must.edu.my
Thu Dec 2 02:31:53 PST 2004

Hello,
I was trying out the tutorial 4 yesterday. Both CntToLedsAndRfm and RfmToLeds were running fine.
For Today, I tried to repeat the steps with other motes. The mote with newly make installed CntToLedsAndRfm seems not sending out any RF signal. While the rest of mote with newly make install RfmToLeds able to receives signal from the mote make installed CntToLedsAndRfm yesterday!
Fyi, I'm using mica2 and made no changes (e.g RF) as in yesterday.

... Could someone let me know what is happening? What should i do??

pls advise.

Next message: [Tinyos-help] CntToLedsAndRfm ain't working today

Ignacio Borraz ignacioborraz at gmail.com
Thu Dec 2 04:10:41 PST 2004

Hello Wong,

It's common that if the antennas of the motes aren't welded, may cause problems of contact. I'm working with mica2 too and I have encountered with similar problems.

Try to push the antenna on his receptacle or move it into it, I think that this is probably your problem.

Hope it helps,
ijborraz

[Tinyos-help] Micro in MicaSBTest2

Rafael Aranha rafael.aranha at tagus.ist.utl.pt
Tue Dec 21 08:40:21 PST 2004

Hello,

Trying MicaSBTest2 I can't get the yellow led to light (and supposing also not the micro), either in the local mote or in another mote nearby. Is there any trick?
I've tried to mangle with muxsel and gainAdjust parameters with no success. I've also looked at the Mic interfaces and implementations at tos/ but there are no comments.

Thank you for you help,
Rafael Aranha

Next message: [Tinyos-help] Micro in MicaSBTest2

Ignacio Borraz ignacioborraz at gmail.com
Tue Dec 21 10:18:06 PST 2004

Hello Rafael,

I had the same problem when I began my work with the sensorboard micasb and the sounder.

The problem is simple; there is an error in the application's code.

The error consists in that the leds interface isn't initalitzed in the MicaSBTest2M.nc code.
The problem is solved only with add the line "call Leds.init()" in the code of StdControl.init() function.

For example as I did it:

```
command result_t StdControl.init() {
state = FALSE;

call Leds.init();

call MicControl.init();
call MicControl.init();
call Mic.muxSel(1);
call Mic.gainAdjust(64);
call PhotoControl.init();
call Sounder.init();
return SUCCESS;
}
```

Hope it helps,
ijborraz

Next message: [Tinyos-help] Micro in MicaSBTest2

Joe Polastre polastre at cs.berkeley.edu
Tue Dec 21 10:59:30 PST 2004

I've committed this fix to CVS. Thanks.

-Joe

[Tinyos-help] surge.Bcast use in JAVA

Till Wimmer wimmer at tkn.tu-berlin.de
Mon Feb 14 13:22:16 PST 2005

Hi all

Is there a example on how to use the net.tinyos.surge.BcastMsg class?

I tried to create a message like this:

```
Message m = new ...
MoteIF moteif = new ...

BcastMsg bmsg = new BcastMsg();
bmsg.dataSet(m, 2);
bmsg.set_seqno(seqNo);
moteif.send(to, bmsg);
```

But then I get the message "unknown AM type for message net.tinyos.surge.BcastMsg".
So what is the idea of BcastMsg... or why is there no AM_TYPE set in tos/lib/Broadcast/Bcast.h?

Thanx
Till

Next message: [Tinyos-help] surge.Bcast use in JAVA

Ignacio Borraz ignacioborraz at gmail.com
Tue Feb 15 03:50:59 PST 2005

Hello Till,

I don't know at all how to resolve your problem, but I think that I can bring some ideas to solve it.

The constructor that you use for create the new object of BcastMsg is defined:

```
    public BcastMsg() {
        super(DEFAULT_MESSAGE_SIZE);
        amTypeSet(AM_TYPE);
```

The method "amTypeSet" is not defined in the class BcastMsg, probably is defined on its superclass Message.
Have you import to your file the package net.tinyos.message???

To the other hand, it could be possible that you need import the header file that defines the AM types for tinyOS.
This file is named AM.h and it's localized in tos/types.

Hope it helps,
ijborraz.

Next message: [Tinyos-help] surge.Bcast use in JAVA

Till Wimmer wimmer at tkn.tu-berlin.de
Tue Feb 15 05:50:41 PST 2005

Hi Ignacio
Thank you for your help.

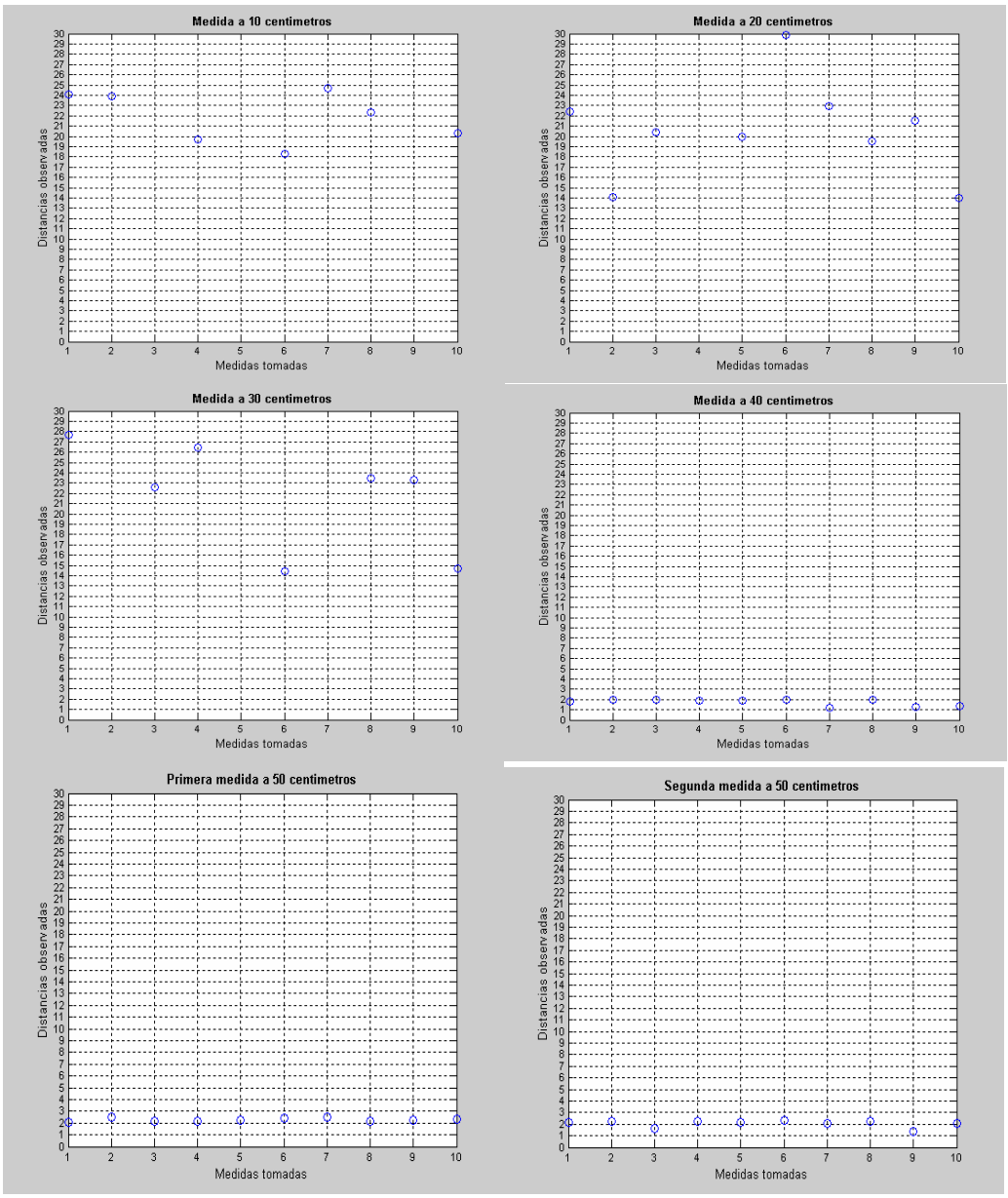
The problem is that AM_TYPE was set to -1 from mig because there is no

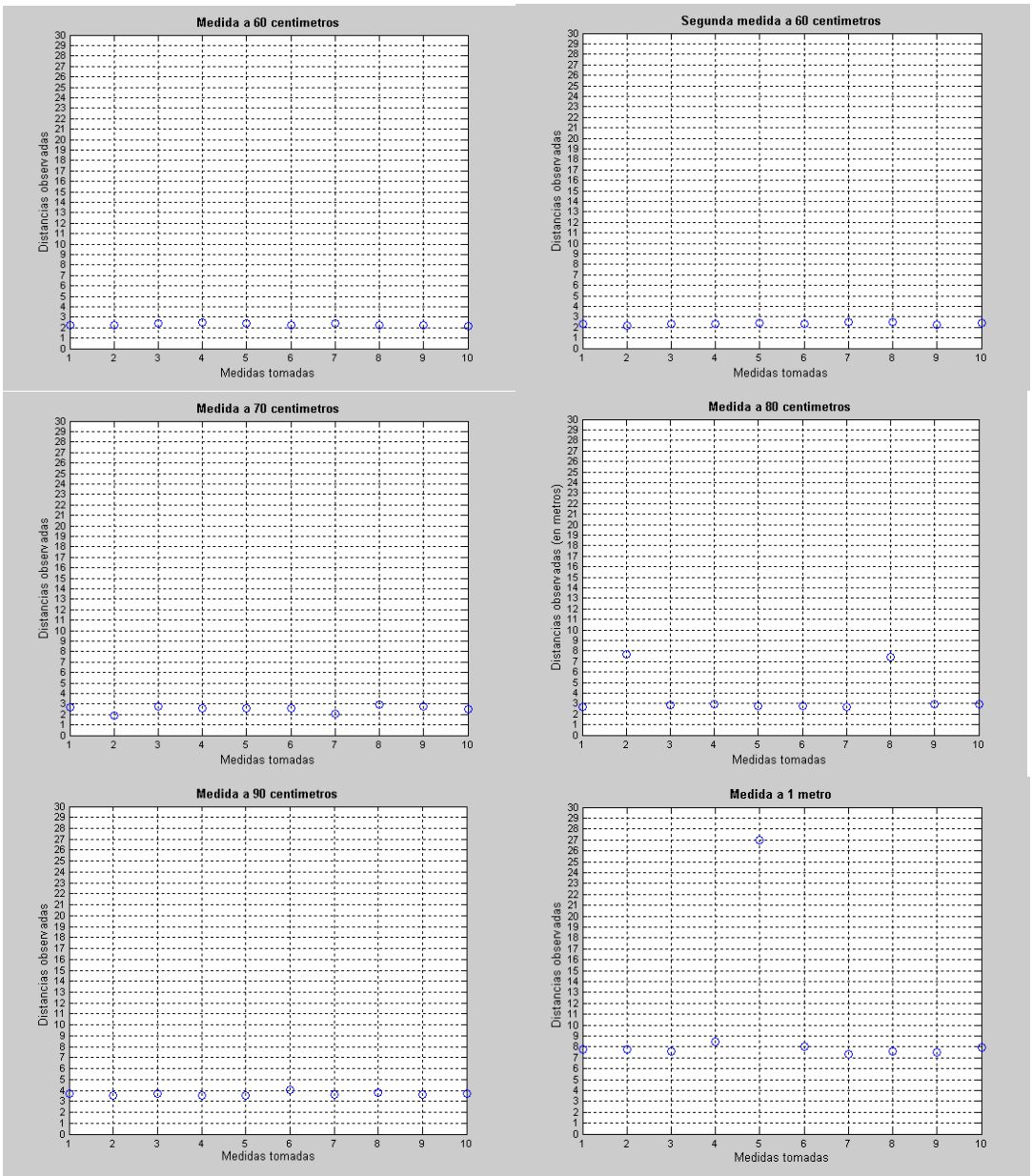
```
enum {
     AM_BCASTMSG= ###;
};
definition in Bcast.h
```

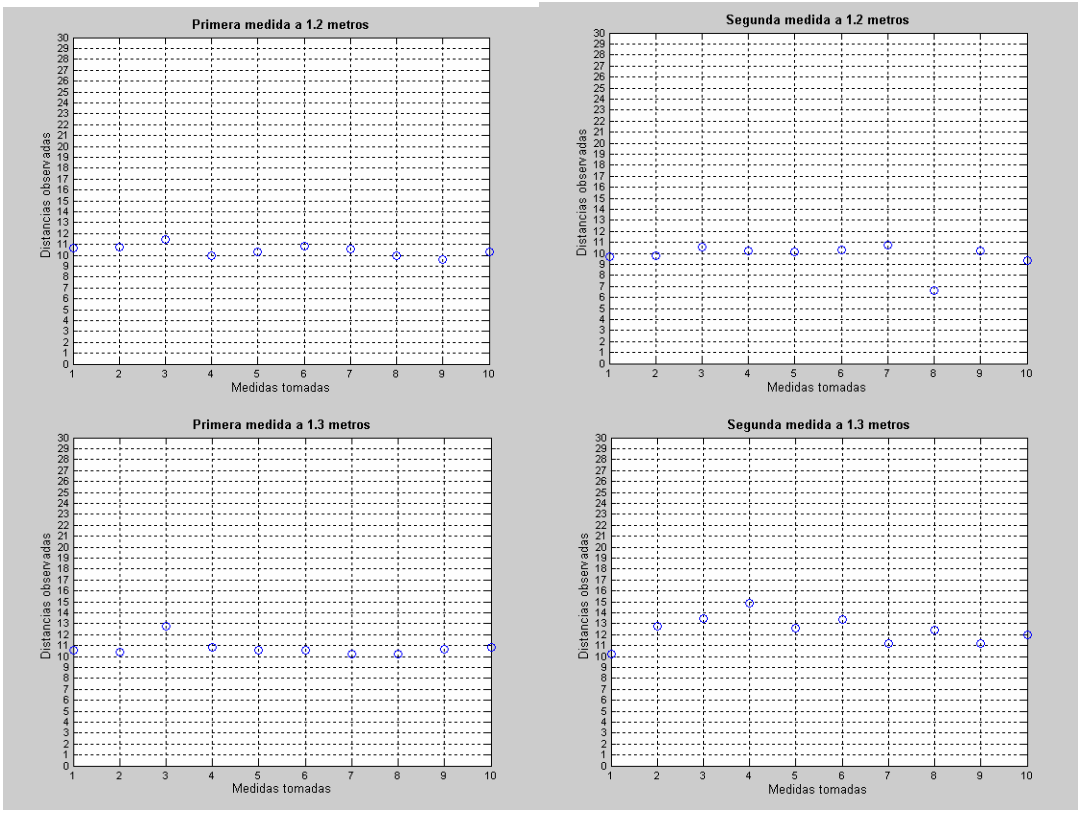
Therefore i wonder what this BcastMsg.class it's good for...

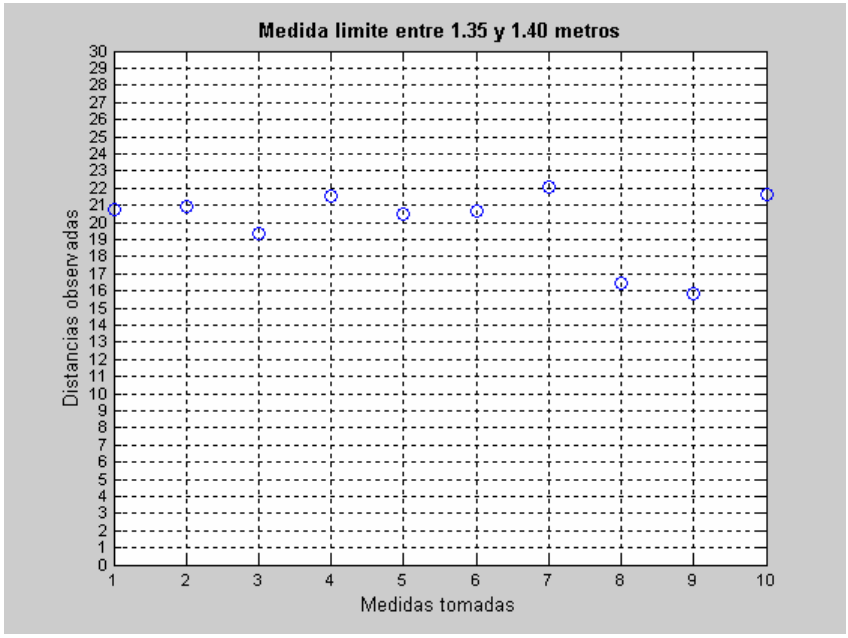
bg
Till

ANEXO 5: Gráficas de resultados




Medida a 60 centimetros
Distancias observadas
Medidas tomadas
Segunda medida a 60 centimetros
Distancias observadas
Medidas tomadas
Medida a 70 centimetros
Distancias observadas
Medidas tomadas
Medida a 80 centimetros
Distancias observadas (en metros)
Medidas tomadas
Medida a 90 centimetros
Distancias observadas
Medidas tomadas
Medida a 1 metro
Distancias observadas
Medidas tomadas


Primera medida a 1.2 metros
Distancias observadas
Medidas tomadas
Segunda medida a 1.2 metros
Distancias observadas
Medidas tomadas
Primera medida a 1.3 metros
Distancias observadas
Medidas tomadas
Segunda medida a 1.3 metros
Distancias observadas
Medidas tomadas


Medida limite entre 1.35 y 1.40 metros
Distancias observadas
Medidas tomadas

BIBLIOGRAFÍA

[1] Sinha, A. and Chandrakasan, A. “Dynamic Power Management in Wireless Sensor Networks,” *IEEE Design Test Comp.*(2001).

[2] Intanagonwiwat, C. Govindan, R. and Estrin, D. “Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks,” *Proc. ACM MobiCom ’00.* 56–67 (2000).

[3] Heinzelman, W.R. Kulik, J. and Balakrishnan, H. “Adaptive Protocols for Information Dissemination in Wireless Sensor Networks,” *Proc. ACM MobiCom ’99*. 174–185 (1999).